Gymnázium, Praha 6, Arabská 14

Obor programování

Ročníková práce

Ivan Merkulov

Snake AI

Duben 2025

Gymnázium, Praha 6, Arabská 14

Arabská 14, Praha 6, 160 00

Ročníková práce

Předmět: Programování

Téma: Snake AI

Školní rok: 2024/2025

Autor: Ivan Merkulov

Třída: 4.E

Vedoucí práce: Mgr. Jan Lána

Třídní učitel: Mgr. Blanka Hniličková

Prohlašuji, že jsem jediným autorem tohoto projektu, všechny citace jsou řádně označené a všechna použitá literatura a další zdroje jsou v práci uvedené. Tímto dle zákona 121/2000 Sb. (tzv. Autorský zákon) ve znění pozdějších předpisů uděluji bezúplatně škole Gymnázium, Praha 6, Arabská 14 oprávnění k výkonu práva na rozmnožování díla (§ 13) a práva na sdělování díla veřejnosti (§ 18) na dobu časově neomezenou a bez omezení územního rozsahu.

V Praze dne 7. dubna 2025 _____

Anotace

Tato práce se zabývá vývojem a implementací hry Snake, ve které had automaticky hraje a dokončí hru tak, že zaplní celé hrací pole, aniž by došlo ke kolizi. Hlavním cílem je nalezení optimální cesty k jablku a minimalizace rizika srážky prostřednictvím různých algoritmů. V rámci práce budou analyzovány a porovnány různé metody plánování pohybu, jako je A* algoritmus a algoritmy pro hledání Hamiltonovské kružnice. Výsledkem práce bude nejen funkční implementace hry, ale také vyhodnocení efektivity jednotlivých přístupů.

Abstract

This work focuses on developing and implementing a Snake game in which the snake plays autonomously and completes the game by filling the entire playing field without colliding. The main objective is to find an optimal path to the apple while minimizing the risk of collision using various algorithms. The work analyzes and compares different pathfinding methods, such as the A* algorithm and algorithms for finding Hamiltonian cycles. The result of this work is a functional implementation of the game together with an evaluation of the efficiency of different approaches.

Anmerkung

Diese Arbeit befasst sich mit der Entwicklung und Implementierung eines Snake-Spiels, in dem die Schlange autonom spielt und das Spiel beendet, indem sie das gesamte Spielfeld füllt, ohne mit sich selbst zu kollidieren. Das Hauptziel ist es, einen optimalen Weg zum Apfel zu finden und gleichzeitig das Kollisionsrisiko durch verschiedene Algorithmen zu minimieren. In dieser Arbeit werden verschiedene Methoden zur Pfadsuche analysiert und verglichen, darunter der A*-Algorithmus und Algorithmen zur Suche nach Hamiltonschen Zyklen. Das Ergebnis dieser Arbeit wird nicht nur eine funktionale Implementierung des Spiels sein, sondern auch eine Bewertung der Effizienz verschiedener Ansätze.

Obsah

1 Úvod

Hra Snake patří mezi klasické počítačové hry, které si získaly popularitu díky své jednoduchosti, ale zároveň rostoucí obtížnosti. Tradičně je had ovládán hráčem, avšak v této práci se zaměřím na vytvoření autonomní verze hry, kde had sám dokáže dohrát hru až do zaplnění celého hracího pole. Aby se had mohl efektivně navigovat herním prostředím, bude využívat algoritmy pro hledání nejoptimálnější cesty k jablku a zároveň se vyhýbat kolizím.

Mezi hlavní metody patří $A*$ algoritmus, který je široce používán pro hledání nejkratších cest, a algoritmy zaměřené na hledání Hamiltonovských kružnic, které umožňují průchod všemi poli bez opakované návštěvy stejného místa. V práci se budu věnovat nejen implementaci samotné hry, ale také porovnání různých přístupů z hlediska jejich efektivity, výpočetní náročnosti a schopnosti hada úspěšně dokončit hru.

1.1 Hra Snake

Hra Snake je jednoduchá arkádová hra, ve které hráč ovládá hada pohybujícího se po hracím poli. Cílem hry je sbírat jablka, která se náhodně objevují na hrací ploše. Po sebrání jablka se had vždy prodlouží o jedno políčko. Hra končí v okamžiku, kdy had narazí do stěny nebo do svého vlastního těla.

Hráč ovládá hada pomocí směrových kláves, čímž mění směr jeho pohybu nahoru, dolů, doleva nebo doprava. Had se pohybuje konstantní rychlostí, a jakmile hráč změní směr, had okamžitě reaguje a pokračuje v pohybu daným směrem. Není možné se otočit o 180 stupňů, což znamená, že had nemůže jít zpět po své vlastní trase.

Obr. 1.1: Hra Snake

V této práci se budu zabývat verzí hry, ve které se had pohybuje v omezeném prostoru s pevnými hranicemi. Hrací pole má obdélníkový tvar a je ohraničeno zdí, která tvoří pevnou bariéru. Jakmile had narazí do této zdi, hra končí. Toto omezení vyžaduje, aby had efektivně plánoval své pohyby a předcházel situacím, kdy by se ocitl v pasti. V této variantě hry neexistuje možnost průchodu skrz okraje obrazovky, což činí navigaci složitější a nutí hada efektivně optimalizovat svůj pohyb.

1.2 Zadání práce

Cílem mé práce je pokusit se o napsání hry Snake, ve které Snake sám dohraje hru (to znamená, že zaplní celý prostor hracího pole, aniž by se zabil). Snake se bude snažit najít optimální cestu k jablku a přitom se bude vyhýbat kolizím. Pomocí různých algoritmů se pokusím o to, aby Snake dokázal najít jablko co nejoptimálněji a pokud možno nejlépe dohrál hru. V práci se budu snažit využít algoritmy na hledání Hamiltonovské kružnice a algoritmus A*. Součástí práce bude porovnání výsledků použitých algoritmů.

1.3 Použité technologie

Práce je napsaná v programovacím jazyce Python, s využitím knihovny Pygame na vykreslování herního pole hry Snake.

1.4 Analýza problému a řešení

Pro úspěšnou implementaci autonomní verze hry Snake je nutné vyřešit několik klíčových problémů. Jedním z hlavních aspektů je reprezentace herního pole a samotného hada. Hrací plocha je modelována jako dvourozměrná mřížka, kde každá buňka může být prázdná, obsahovat tělo hada nebo jablko. Had je reprezentován jako seznam souřadnic jeho těla, kde první prvek odpovídá hlavě hada.

Aby bylo možné efektivně plánovat pohyb hada, problém navigace lze chápat jako hledání cesty v grafu, kde uzly představují jednotlivé buňky hracího pole a hrany odpovídají možným tahům. Hledání nejkratší cesty probíhá mezi hlavou hada a jablkem, což snižuje výpočetní náročnost oproti sledování celého těla. Hlavní výzvou je nejen nalezení efektivní trasy k jablku, ale také zajištění toho, aby had nezablokoval svůj vlastní pohyb a měl dostatečný prostor pro další růst. Kromě toho musí být strategie hada navržena tak, aby umožnila zaplnění celé hrací plochy, tedy úspěšné dokončení hry.

2 Implementace grafu

Hra Snake probíhá na čtvercovém hracím poli, kde se had pohybuje a snaží se sníst jablka, která se na něm objevují. Had se pohybuje postupně po jednotlivých políčkách, přičemž musí optimalizovat svou trasu tak, aby dosáhl jablka co nejefektivněji a zároveň se neuvěznil ve vlastní stopě. Abychom mohli efektivně řídit jeho pohyb a hledat nejkratší cesty, můžeme tento problém převést na graf.

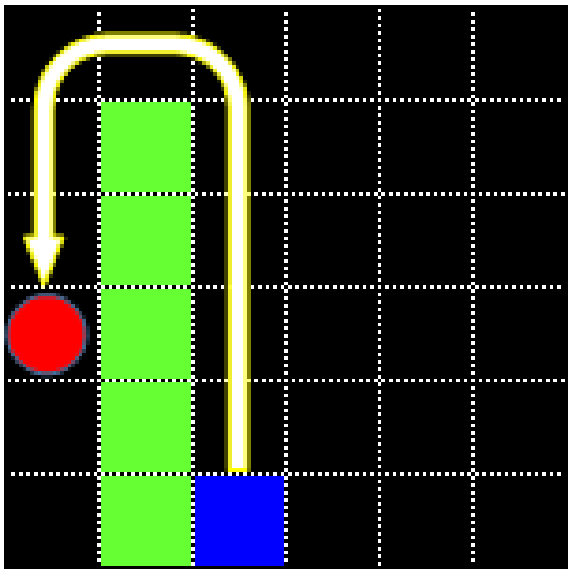
Hrací pole obsahuje $n \times n$ políček, kde n je počet políček na jedné straně. Takové hrací pole dokážeme jednoduše reprezentovat jako čtvercový graf, což je graf, jehož vrcholy jsou uspořádány do pravidelné pravoúhlé mřížky o n^2 vrcholech. Každý vrchol bude reprezentovat jedno políčko v hracím poli. Hrany grafu pak reprezentují možné pohyby hada mezi sousedními políčky. Každý vrchol má různý počet hran v závislosti na své poloze v mřížce. Do vrcholů reprezentující rohová políčka povedou dvě hrany. Uzly, které představují políčka na okraji hracího pole, budou napojeny třemi hranami a do vrcholů, které jsou uprostřed, vedou čtyři hrany.

Tento graf lze reprezentovat dvěma způsoby: staticky, kdy se celý graf předem vygeneruje a uloží, nebo dynamicky, kdy se vrcholy a hrany vytvářejí pouze podle potřeby.

2.1 Statická reprezentace

V statické reprezentaci grafu si vygenerujeme najednou graf pro celé hrací pole a celý graf si uložíme do paměti. Tento graf nebudeme následně během hry nijak měnit. Pokud chceme provést změny v grafu, musíme celý graf smazat a vygenerovat nový graf, reflektující všechny požadované změny.

Taková reprezentace je výhodná, pokud graf využíváme jako reprezentaci nějaké situace, která se nemění, tedy že se nemění graf, se kterým pracujeme. Např., když potřebujeme najít nejkratší cestu na mapách, stejnou mapu v tomto případě lze využít i opakovaně. Nejprve najdeme například nejkratší cestu domu a pokud budeme sledovat nalezenou



Obr. 2.1: Nejkratší cesta od hlavy k jablku - statický graf, tělo hada - zelené čtverečky, hlava hada - modrý čtvereček, jablko - červené kolečko, šipka - nejkratší cesta od hlavy hada k jablku

trasu, cesta se v průběhu pohybu měnit nebude. Následně můžeme využít stejného grafu (mapy), abychom našli cestu z domova do školy, ze školy na kroužek atd.

Výhodou takového přístupu tedy je, že generujeme pouze jeden graf. Nevýhodou je naopak potřeba generovat graf celý, tj. se všemi vrcholy a hranami. Když totiž hledáme nejkratší cestu (například pomocí algoritmu $A*$) z bodu A do bodu B v grafu, tak některé vrcholy pro nalezení nejkratší cesty nejen že nevyužijeme, ale ani je nebudeme kontrolovat. Při malém grafu to úplně nevadí, ale když je graf větší, může zabírat opravdu mnoho paměti.

Uveďme si nyní, jak se dá statická reprezentace grafu použít v prostředí hry Snake při hledání nejkratší cesty od hlavy hada k jablku.

Vygenerujeme si graf, kde budou propojeny všechny vrcholy tak, jak již bylo zmíněno v úvodu této kapitoly, s tím rozdílem, že do vrcholů grafu, ve kterých se nenachází tělo hada, nepovedou žádné hrany z okolních políček. Z těchto vrcholů povedou hrany pouze po směru pohybu hada do dalších částí jeho těla, tedy postupně propojíme orientovanými hranami jednotlivé části těla hada od ocasu až k hlavě.

Tím docílíme toho, že algoritmus pro hledání nejkratší cesty se vyhne takovým cestám, ve kterých by had narazil sám do sebe. Algoritmus pro hledání nejkratší cesty tedy najde cestu podobnou jako na Obrázku 2.1. Vidíme, že nalezená cesta závisí pouze na počáteční situaci a algoritmus nebere vůbec v potaz, že se had bude hýbat (tedy, že v průběhu se

může uvolnit prostor pro ještě kratší cestu).

Po vytvoření grafu algoritmus pro hledání nejkratší cesty najde nejkratší cestu s ohledem na počáteční situaci. To zároveň znamená, že pokud tělo hada momentálně zakrývá přímou cestu k jablku, tak ve statickém grafu to bude vnímáno tak, že bude cesta k jablku zakrytá pořád.

Ve hře Snake se had ovšem hýbe. Proto, pokud bude sledovat cestu nalezenou ve statickém grafu, bude ke konci obcházet překážku, která již reálně neexistuje. Při takovém přístupu budeme muset navíc po každém sebraném jablku vygenerovat nový graf, protože had neustále mění svoji pozici. Jak již bylo zmíněno, generování nového grafu je náročná operace a zejména při velkém grafu by trvala velmi dlouho. Proto statická reprezentace grafu pro simulaci hry Snake není úplně vhodná.

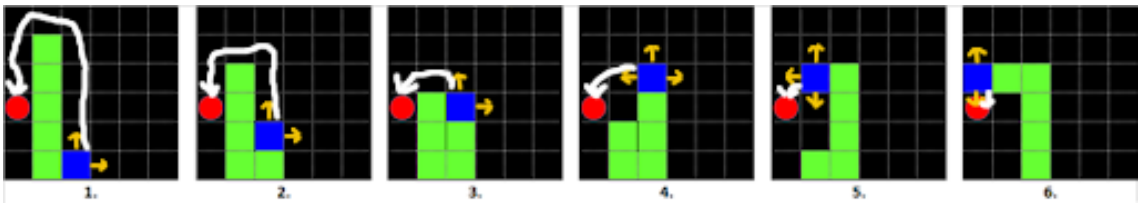
2.2 Dynamická reprezentace

V dynamické reprezentaci není nutné předem generovat celý graf, ale vrcholy se vytvářejí a prozkoumávají postupně během hledání nejkratší cesty. Každý vrchol obsahuje informace o svém předchozím stavu, tedy odkud jsme se do něj dostali, což umožňuje zpětně rekonstruovat nalezenou cestu.

Tento proces probíhá iterativně – na začátku máme pouze výchozí stav (pozici hlavy hada) a postupně rozšiřujeme stavový prostor přidáváním sousedních vrcholů (možných pohybů hada). Každý nový stav odpovídá novému uspořádání herního pole po provedení daného tahu. Pokud narazíme na cílový stav (pozici jablka), můžeme zpětným sledováním předchozích stavů rekonstruovat nejkratší cestu.

Dynamická reprezentace tímto způsobem odpovídá principu prohledávání do šířky (BFS), kde se nejprve prozkoumávají všechny vrcholy v aktuální úrovni a teprve poté se přechází na další. BFS zajišťuje, že první nalezená cesta k cíli je zároveň nejkratší (pokud všechny tahy mají stejnou cenu), což je výhodné právě pro plánování pohybu hada. Hlavní rozdíl oproti statické reprezentaci je v tom, že zde nevytváříme celý graf dopředu, ale rozšiřujeme ho postupně podle potřeby. Tím se výrazně šetří paměť a umožňuje efektivní vyhledávání i ve větších prostředích. Podrobněji si to vysvětlíme na stejném příkladu jako při generování

statického grafu (2.1).



Obr. 2.2: Nejkratší cesta od hlavy k jablku - dynamický graf, tělo hada - zelené čtverečky, hlava hada - modrý čtvereček, jablko - červené kolečko, žluté šipky - části grafu, které se aktuálně prozkoumávají pro hledání cesty, bílé šipky - aktuální nejkratší cesta od hlavy hada k jablku

Máme dané pouze vrcholy reprezentující tělo hada a vrchol jablka. V prvním kroku na Obrázku 2.2 vidíme, že se had v dalším kroku dokáže pohnout pouze před sebe (rovně) nebo nahoru (doleva). To jsou vrcholy, které budeme prozkoumávat dál a také vrcholy, které vygenerujeme. Vygenerujeme je ovšem již pro situaci, která nastane, až se had na dané políčko posune. To znamená, že například pro možnost, kdy se had pohne nahoru (doleva), se každá část jeho těla posune o jedno políčko dopředu, na předcházející místo navazující části těla.

Druhý krok z Obrázku 2.2 tedy popisuje pozici hada, v případě, že se rozhodl pohnout v prvním kroku nahoru (doleva). Vidíme, že z této pozice se had může posunout buď doprava nebo nahoru (rovně). Postup z prvního kroku opakujeme a generujeme tak další možné stavy, dokud nenajdeme jablko. V každém stavu vygenerujeme pouze tělo hada a uložíme si k tomu i předchozí stav (stav, ze kterého vznikl aktuální stav).

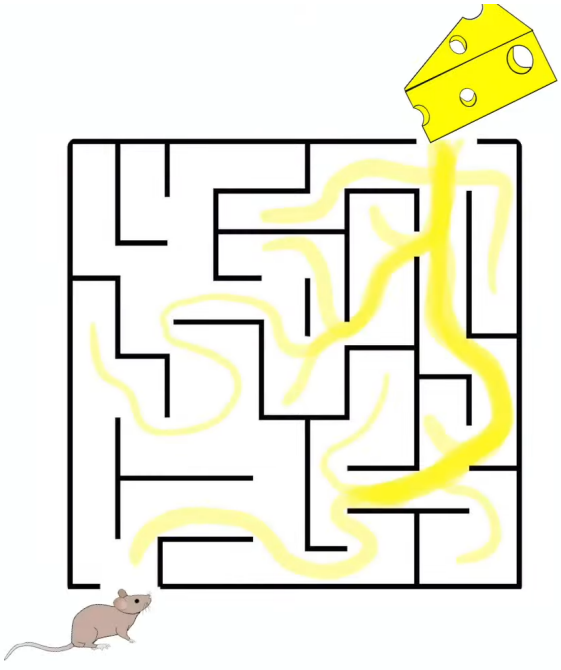
Jak je vidět na Obrázku 2.2, tento způsob reprezentace grafu nám umožňuje při hledání nejkratší cesty počítat i s pohybem hada v čase a díky tomu najít optimálnější cestu. Také nemusíme pokaždé, když hledáme novou cestu, vytvářet znovu celý graf, ale stačí nám vygenerovat pouze dílčí stavy, což je mnohem výhodnější nejen, co se týče paměti, ale i samotné rychlosti generování.

3 Algoritmus A*

Další možností, jak dohrát hru Snake, je stále hledat nejkratší možnou cestu od hlavy hada k jablku. Existuje mnoho algoritmů, kterými dokážeme najít nejkratší cestu z bodu A do bodu B, ale jedním z nejlepších je algoritmus A* (A - star). Tento algoritmus funguje na principu Dijkstrova algoritmu [1], ale ke zrychlení výpočtů využívá heuristiku.

3.1 Heuristika

Proto abychom si lépe představili, co to heuristika je, uvedeme si příklad bludiště, kde na startu stojí myš a snaží se dostat na konec bludiště k sýru. Když se snaží myš dostat do cíle, tak může zvolit jakoukoli z možných cest. Tyto cesty mohou vést v lepším případě blíže k cíli, v horším ji zavedou naopak ještě dál od cíle, než byla na startu. Kdyby však cítila zápach sýru, věděla by vždy, jestli cítí zápach sýru méně, a tím pádem se vzdaluje, nebo jestli se zápach zesiluje a ona se k cíli naopak přibližuje (viz Obrázek 3.1).



Obr. 3.1: Myš a bludiště se sýrem (žluté čáry udávají intenzitu zápachu sýra)

Díky tomu nemusí zkoušet cesty, které ji silně oddalují od cíle. Stále sice může vybrat cestu, která není správná, ale bude moci rychle rozpoznat, že tato cesta je špatná a vrátí se na tu správnou. V případě myši a sýru si můžeme heuristiku představit jako vůni, kterou myš cítí. Heuristika nám udává přibližný odhad toho, jak daleko jsme od cíle.

V hracím poli pro hru Snake můžeme heuristiku reprezentovat jako Manhattanskou vzdálenost, což je součet absolutních rozdílů souřadnic mezi dvěma body na mřížce.

Matematicky ji lze vyjádřit jako:

$$d = |x_1 - x_2| + |y_1 - y_2|,$$

kde (x_1, y_1) jsou souřadnice hlavy hada a (x_2, y_2) souřadnice jablka. Taková heuristika je pro náš problém vhodná, protože had se dokáže pohybovat pouze nahoru, dolů, doleva a doprava.

3.2 Popis algoritmu

Jak již bylo zmíněno v úvodu kapitoly, algoritmus funguje na principu Dijkstrova algoritmu, má ale efektivnější výběr dalších vrcholů na zkoumání, ke kterému využívá heuristiku. V praxi se toho dosahuje prioritní frontou, ve které se vrcholy (V) řadí podle hodnoty $f(V)$.

$f(V)$ definujeme následně:

$$f(V) = g(V) + h(V),$$

kde $g(V)$ je vzdálenost od počátečního vrcholu k mu vrcholu V a $h(V)$ je heuristická vzdálenost od vrcholu V k cíli.

Postup samotného algoritmu je následující (na jeho implementaci v pythonu lze nahlédnout v Ukázce kódu 3.1):

1. Přidá počáteční vrchol do prioritní fronty s jeho $f(V)$.
2. Dokud fronta není prázdná, vrchol s nejnižší hodnotou $f(V)$ bude z fronty vyřazen.
3. Pokud je tento vrchol cíl, algoritmus se ukončí a vrátí zjištěnou cestu.
4. V opačném případě se rozbalí uzel (najdou se všechny jeho sousedy) a vypočítají se $g(V), h(V)$ a $f(V)$ pro každého souseda. Každý soused se přidá do fronty, pokud

tam ještě není, nebo pokud se k tomuto sousedovi nenajde lepší cesta.

5. Cyklus se opakuje, dokud se nedojde k cíli nebo ve frontě už nezbyli žádné další uzly, což znamená, že neexistuje cesta od startu k vrcholu.

```python
def a_star(graph, start, goal):
open_set = []
heapq.heappush(open_set, (0, start))
came_from = {}
g_score = {node: float('inf') for node in graph}
g_score[start] = 0
f_score = {node: float('inf') for node in graph}
f_score[start] = heuristic(start, goal)

while open_set:
    _, current = heapq.heappop(open_set)
    if current == goal:
        return reconstruct_path(came_from, current)

    for neighbor, cost in graph[current].items():
        tentative_g_score = g_score[current] + cost
        if tentative_g_score < g_score[neighbor]:
            came_from[neighbor] = current
            g_score[neighbor] = tentative_g_score
            f_score[neighbor] = g_score[neighbor] + heuristic(neighbor, goal)
            heapq.heappush(open_set, (f_score[neighbor], neighbor))

return None
```

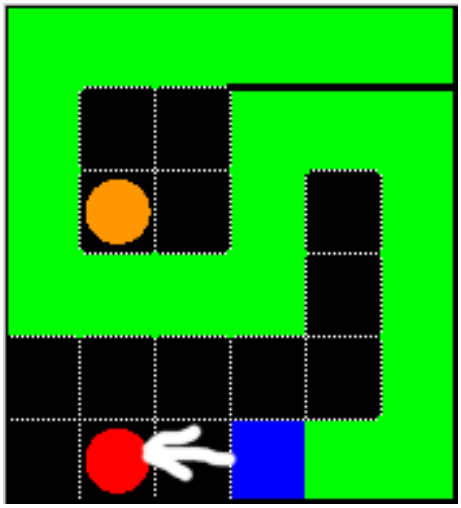
Ukázka kódu 3.1: Implementace A* v pythonu

3.3 Nesprávnost řešení hry

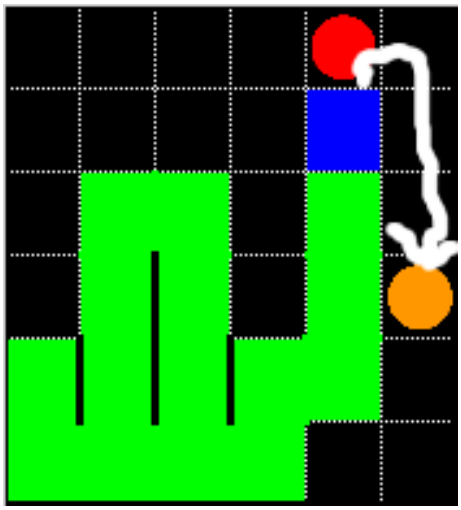
Použití algoritmu pro hledání nejkratší cesty od hlavy hada k jablku je sice z časového hlediska rozhodně rychlé, ale nikdy nedokáže zaplnit celé hrací pole, neboli úspěšně dohrát hru. Když se had bude postupně zvětšovat, bude na hracím poli méně volných políček a některá můžou být tělem hada kompletně odříznutá, kvůli čemuž had naboura do sebe a zemře, aniž by zaplnil celé hrací pole.

Pro objasnění si představme následující situaci. Had už je docela veliký a míří sníst další jablko. Cesta, kterou zvolil, oddělila část volných políček od ostatních. Nyní, když sní jablko a další se vygeneruje právě v tom odděleném prostoru, který v předchozích krocích

vytvořil, neuvidí ho, neboli neexistuje žádná cesta od hlavy hada k jablku (viz Obrázek 3.2). Další případ, na kterém tento způsob kolabuje, je, když si had během pohybu k jablku odstřihne únikovou cestu, neboli cestu, po které by se mohl pohybovat zpátky poté, co jablko sní(viz Obrázek 3.3).



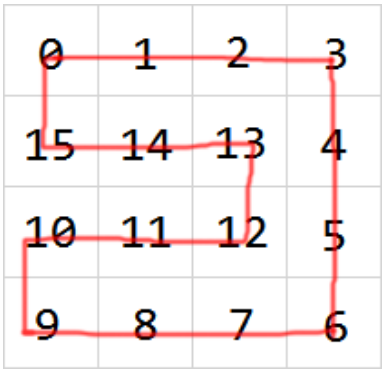
Obr. 3.2: Had, který nemůže najít cestu k dalšímu vygenerovanému jablku, zelené čtverečky - tělo hada, modrý čtvereček - hlava hada, červené kolečko - jablko, které vidí aktuálně, oranžové kolečko - jablko, které uvidí poté co sebere červené jablko, bílá šipka - cesta, kterou had projde při hledání obou jablek



Obr. 3.3: Had, který si zablokoval únikovou cestu, zelené čtverečky - tělo hada, modrý čtvereček - hlava hada, červené kolečko - jablko, které vidí aktuálně, oranžové kolečko - jablko, které uvidí poté co sebere červené jablko, bílá šipka - cesta, kterou had projde při hledání obou jablek

4 Hamiltonovská kružnice

Jednou z možností, jak docílit úspěšného dokončení hry Snake, je donutit hada v každé iteraci projít každé políčko právě jednou (viz Obrázek 4.1). Takovým způsobem zamezíme tomu, že had narazí sám do sebe, a zároveň, že si had při prodlužování nezamezí úplně přístup k nějaké části hracího pole, tedy že i po maximálním prodloužení zůstane každé políčko dosažitelné. Takové cestě se v grafové teorii říká Hamiltonovská kružnice [2]. Nalezení Hamiltonovské kružnice je ovšem výpočetně náročný problém (jinak známý jako problém obchodního cestujícího), konkrétně patří do třídy NP- úplných problémů. To znamená, že časová složitost nalezení Hamiltonovské kružnice je $O(n!)$ a složitost ověření správnosti řešení je také $O(n!)$.



Obr. 4.1: Příklad hamiltonovské kružnice pro hrací pole 4×4

4.1 Třídy složitostí

Jedním z nejslavnějších a nejtěžších problémů, který v informatice existuje, je problém P versus NP. P označuje množinu všech algoritmických problémů, které je možné vyřešit deterministickým Turingovým strojem v polynomiálním čase. Polynomem [3] rozumíme matematický výraz ve tvaru:

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

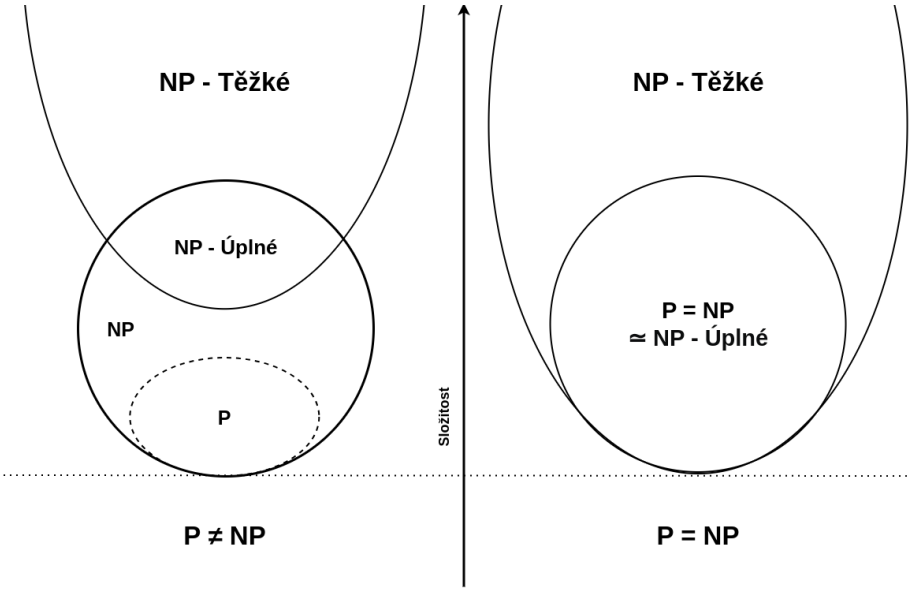
kde $a_0, a_1, \ldots, a_{n-1}, a_n$ jsou reálné koeficienty a n je nezáporné celé číslo, které určuje stupeň polynomu. Příkladem polynomu může být: $4x^3 + 5x^2 - 8$.

Vyřešit problém v polynomiálním čase znamená, že doba, kterou algoritmus potřebuje k vyřešení problému, je omezena polynomem závislém na velikosti vstupu. Protože Landauova notace [4] (známá také jako notace velké O) zanedbává konstantní koeficienty a zaměřuje se pouze na nejvyšší prvek polynomu, můžeme říci, že algoritmus řešící problém v polynomiálním čase má časovou složitost $O(n^k)$, kde n je velikost vstupu a k je konstanta označující stupeň polynomu. To znamená, že doba běhu algoritmu roste nejvýše polynomiálně vzhledem k velikosti vstupu. To znamená, že pro dostatečně velké n je růstová rychlost takové funkce pomalejší než exponenciální $O(2^n)$ nebo faktoriálové $O(n!)$ složitosti, což činí problémy z třídy P prakticky řešitelnými. Pro problémy z třídy P vždy existuje řešení, které problém řeší v polynomiálním čase, a existuje i způsob, jak v polynomiálním čase ověřit správnost výsledku.

Problémy třídy NP definujeme jako ty, které jsou v polynomiálním čase řešitelné nedeterministickým Turingovým strojem, jehož myšlenka je pouze hypotetická. Zjednodušeně můžeme říci, že NP problémy jsou takové, jejichž správnost řešení dokážeme ověřit v polynomiálním čase, ale nedokážeme přesně určit, zdali lze řešení v polynomiálním čase najít.

Všechny problémy v NP dokážeme redukovat na jiné problémy ve třídě NP, to znamená, že pokud bychom dokázali najít řešení v polynomiálním čase pro jeden problém z NP, dokázali bychom tím, že všechny problémy v NP lze řešit v polynomiálním čase. To by ale také znamenalo, že $P = NP$. Většina odborníků se ovšem domnívá, že takový efektivní algoritmus neexistuje, a proto je většina přesvědčena o tom, že rovnost neplatí, tedy že $P \neq NP$.

Třídy složitosti můžeme dále dělit na NP- úplné a na NP- těžké problémy [5]. NP- těžké problémy popisují třídu, ve které jsou problémy těžké alespoň jako všechny NP složité problémy. To znamená, že bude obsahovat problémy, které budou určitě těžší než NP. Třída NP- úplných problémů obsahuje úlohy, které jsou určitě těžké jako všechny NP problémy, ale nejsou těžší. To znamená, že tato třída leží přímo mezi NP a NP- těžkých problémů (hierarchie tříd je zobrazena na Obrázku 4.2).



Obr. 4.2: Hierarchie tříd složitosti

4.2 Neúplné řešení

Jak bylo řečeno dříve, hledání Hamiltonovské kružnice patří mezi NP- úplné problémy, tedy že pro obecný graf neumíme najít Hamiltonovskou kružnici efektivně. Jsme schopni nalézt takovou kružnici v časové složitosti $O(n!)$, to znamená, že pro velmi malá n dokážeme najít kružnici celkem rychle, ale třeba už pro $n = 17$ by výpočet trval přibližně 4 dny.

Graf používaný ve hře Snake je čtvercový. Omezení daná z jeho definice nám usnadní úlohu hledání kružnice, protože problém již nemusíme řešit pro obecný graf, který má mnohem více hran. Pro čtvercový graf jde relativně snadno najít Hamiltonovská kružnice, pomocí algoritmu od Johna Tapsella [6].

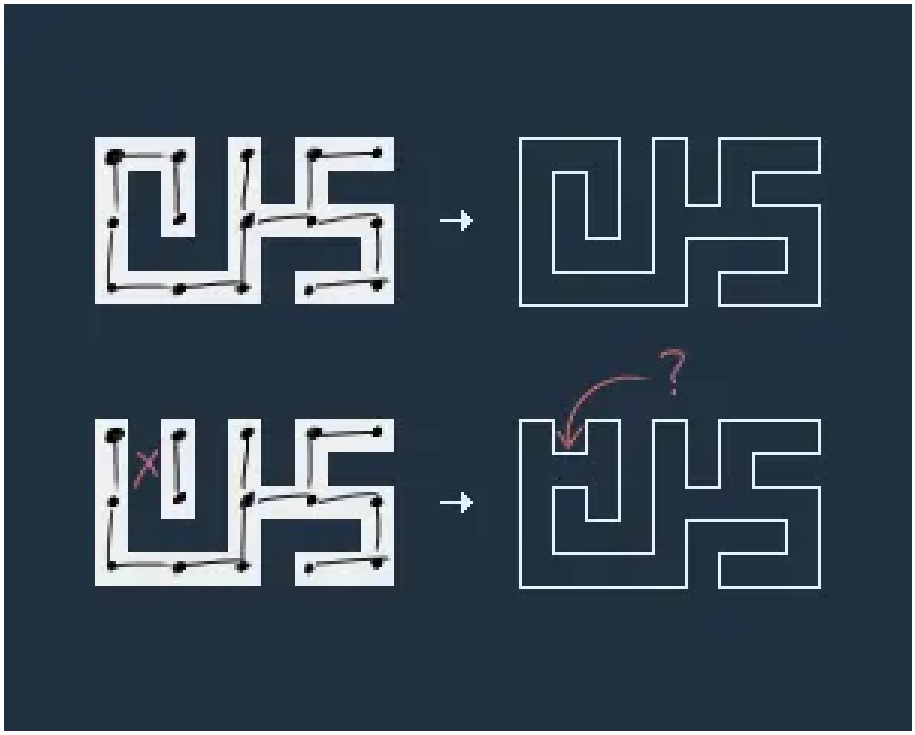
4.2.1 Algoritmus poloviční kostry grafu

Nejdůležitější myšlenka algoritmu pro hledání Hamiltonovské kružnice ve čtvercovém grafu se zakládá na kostře grafu (Obrázek 4.3a). Pokud se pozorně podíváme na kostru nějakého čtvercového grafu, zjistíme, že obvod kostry je vlastně Hamiltonovská kružnice pro graf dvakrát větší (Obrázek 4.3b). To znamená, že pro nalezení kružnice potřebujeme vygenerovat kostru pro graf s dvakrát menšími rozměry než má graf, ve kterém se snažíme najít kružnici.

(a) Kostry grafu

(b) Transformace z kostry na Hamiltonovskou kružnici

Obr. 4.3: Kostry grafu a jejich transformace



Obr. 4.4: Kostra grafu, která neumožňuje vygenerovat všechny možné Hamiltonovské kružnice, ale pouze jejich podmnožinu

Nalezení kostry velmi ulehčuje hledání Hamiltonovské kružnice ve čtvercovém grafu, ale bohužel můžeme tímto způsobem vygenerovat pouze podmnožinu všech možných Hamiltonovských kružnic (Obrázek 4.4). Zvětšováním rozměrů grafu $n \times m$, ve kterém hledáme kostru, se zároveň omezuje i velikost grafu, ve kterém následně hledáme Hamiltonovskou kružnici – ta je určena rozměry $2n \times 2m$.

Pro úspěšné vyřešení hry Snake ovšem nepotřebujeme najít všechny možné Hamiltonovské kružnice, stačí nám alespoň jedna. Proto můžeme přístup založený na hledání kostry grafu bez obav využít.

4.2.2 Nález Hamiltonovské kružnice

Díky algoritmu popsanému v předchozí sekci dokážeme snadno najít Hamiltonovskou kružnici, jako obvod kostry dvakrát menšího grafu. Jak ale v praxi vytvoříme graf jako obvod kostry?

První, co si musíme uvědomit, je, že každý vrchol z menšího grafu má okolo sebe vždy čtyři vrcholy většího grafu. Ke každému vrcholu musíme tedy správně přiřadit čtyři vrcholy většího grafu. Zkusme přemýšlet nad konkrétním příkladem.

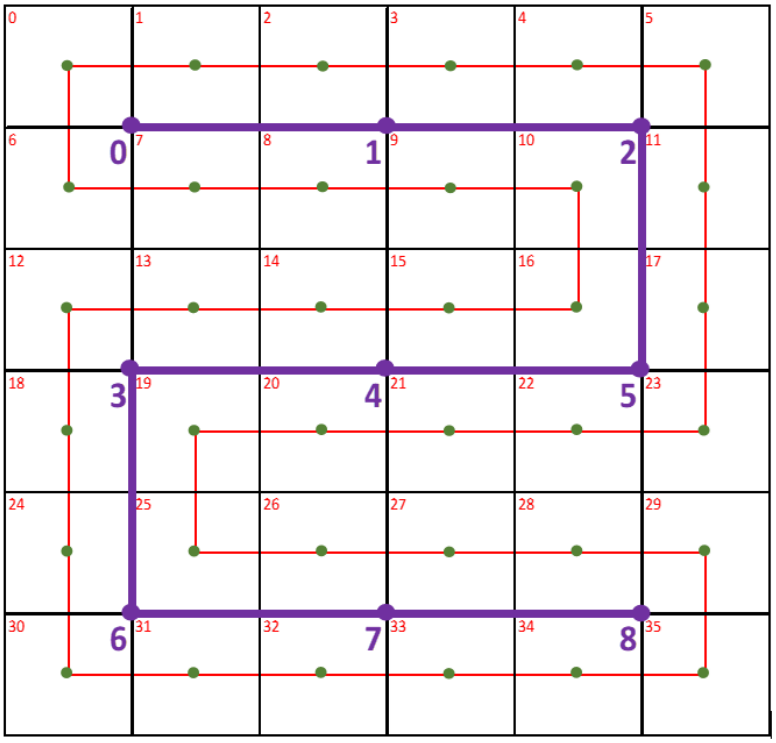
Graf, ve kterém budeme hledat kostru, bude mít rozměry 3×3, tím pádem graf, ve kterém se budeme snažit najít kružnici, má rozměry 6×6. Vrcholy budeme číslovat od nuly, menší graf bude mít vrcholy 0-8, větší graf 0-35. Z Obrázku 4.5 je patrné, že stačí zjistit číslo vrcholu, který je od vrcholu menšího grafu umístěn vlevo nahoře, protože ostatní tři vrcholy dokážeme zjistit přičtením fixních konstant. Jak se konkrétně přiřadí sousední vrcholy, ilustrujeme pomocí metody `neighbor_nodes` 4.1.

Když už máme přiřazené sousední vrcholy, stačí zjistit, které z nich potřebujeme spojit, abychom získali Hamiltonovskou kružnici. Z Obrázku 4.5 vidíme, že rozhodně můžeme vždy spojit vrcholy, které se nacházejí v rohu hracího pole (v našem případě $[6, 0]$ a $[0, 1]$, nebo $[24, 30]$ a $[30, 31]$ atd.). Další hrany, o kterých určitě víme, že je můžeme vytvořit, jsou hrany mezi vrcholy, které se nacházejí na krajích hracího pole a zároveň sousedí s vrcholy přiřazenými ke stejnému vrcholu z menšího grafu. To jsou např.: $[12, 18]$, $[2, 3]$ nebo $[17, 23]$.

```
    def neighbor_nodes(self):
        nodes_for_smaller_nodes = {}
        for x in range(self.vertices):
            left_up = ((x // self.number_of_nodes_on_side)
            * 4 * self.number_of_nodes_on_side)
            + ((x % self.number_of_nodes_on_side) * 2)
            right_up = left_up + 1
            left_down = left_up + (self.number_of_nodes_on_side * 2)
            right_down = left_down + 1
            # Save neighbor nodes into the dictionary
            nodes_for_smaller_nodes[x] = {
                "left_up": left_up,
                "right_up": right_up,
                "left_down": left_down,
                "right_down": right_down
            }
        return nodes_for_smaller_nodes
```

Ukázka kódu 4.1: Určení sousedních vrcholů většího grafu

O ostatních hranách již nemůžeme rozhodnout tak jednoduše bez využití kostry menšího grafu. Vezměme tedy např. hranu z menšího grafu $[0, 1]$, vidíme, že pokud chceme utvořit obrys této hrany, potřebujeme spojit vrcholy nad hranou $[1, 2]$ a pod hranou $[7, 8]$. Tyto hrany jsou rovnoběžné k hraně $[0, 1]$ malého grafu. Kdyby hrana $[0, 1]$ mezi vrcholy malého grafu nebyla, museli bychom spojit sousední vrcholy bodů 0 a 1 ve směru kolmém na hypotetickou hranu $[0, 1]$. To znamená hrany $[1, 7]$ a $[2, 8]$. Takovým způsobem pokračujeme ve stavbě celého grafu. Procházíme všechny vrcholy malého grafu, které tvoří kostru. Pokud existuje hrana mezi dvěma vrcholy v menším grafu, spojíme odpovídající sousední vrcholy v rozšířeném grafu rovnoběžně se směrem této hrany. Pokud v menším grafu hrana neexistuje, spojíme odpovídající sousední vrcholy ve větším grafu ve směru kolmém na směr, ve kterém by tato hrana vedla.



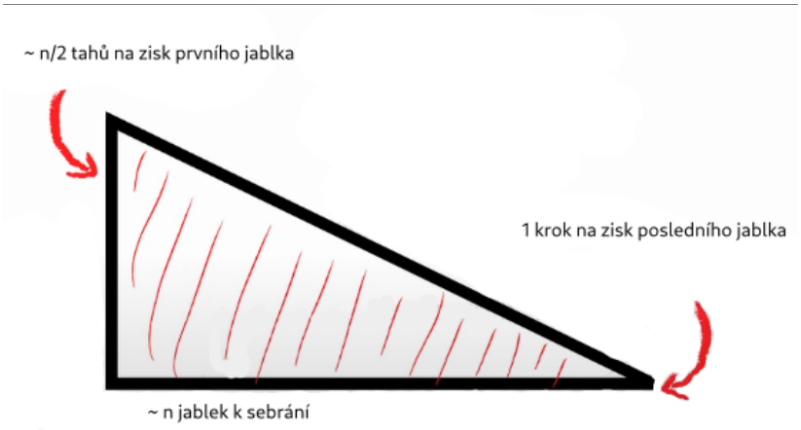
Obr. 4.5: Kostra malého grafu, Hamiltonovská kružnice velkého grafu, sousední vrcholy (zelené)

4.3 Rychlost řešení hry

Použití Hamiltonovské kružnice pro úspěšné dokončení hry Snake funguje vždy [7]. V porovnání s jinými metodami, které budou popsány dále v textu, je ovšem nesmírně pomalé (porovnání v kapitole 6). Cílem této práce nebylo pouze úspěšné dokončení hry, ale její dokončení v co nejkratším čase.

Při tomto přístupu k řešení hry se například vůbec neuvažuje nad tím, kde se objeví další jablko. To znamená, že na začátku hry, kdy je had ještě velmi malý, je průměrná vzdálenost, kterou had musí urazit k jablku, polovina hracího pole, neboli $\frac{n}{2}$ všech vrcholů v grafu. V polovině hry, kdy had zabírá nějaký prostor, se už jablko musí objevit blíže k hadovi a na konci hry, když had zabírá skoro celé hrací pole, se bude jablko objevovat přímo před ním. K poslednímu jablku, které had sebere, urazí cestu dlouhou jedno políčko.

Tím pádem dokážeme vypočítat průměrný počet tahů na hru při použití Hamiltonovské kružnice pro dokončení hry. Počet tahů vyjádříme následujícím vztahem:



Obr. 4.6: Výpočet přibližného počtu tahů

$$\frac{\frac{n}{2} \cdot n}{2}$$

kde n je počet vrcholů v grafu (počet políček v herním poli) [8]. Závislost můžeme nahlédnout také na Obrázku 4.6.

5 Řešení

V kapitolách 4 a 3 jsme uvedli možné přístupy použitelné pro simulaci hry Snake a k dosažení jejího dokončení. Oba přístupy ale mají své nedostatky. V této kapitole vysvětlím své optimálnější řešení problému, ke kterému jsem využil algoritmus Johna Tapsella [6].

5.1 Popis algoritmu

Algoritmus využívá Hamiltonovské kružnice, protože jak již víme, sledováním takové kružnice umožníme hadovi hru úspěšně dohrát, aniž by v průběhu narazil sám do sebe. Po vygenerování kružnice jí stále budeme následovat, ale s tím rozdílem, že si občas cestu zkrátíme, abychom sebrali jablko rychleji a následně se opět vrátili na kružnici. Tím bychom dokázali hru ve fázi, kdy je had ještě krátký, výrazně urychlit.

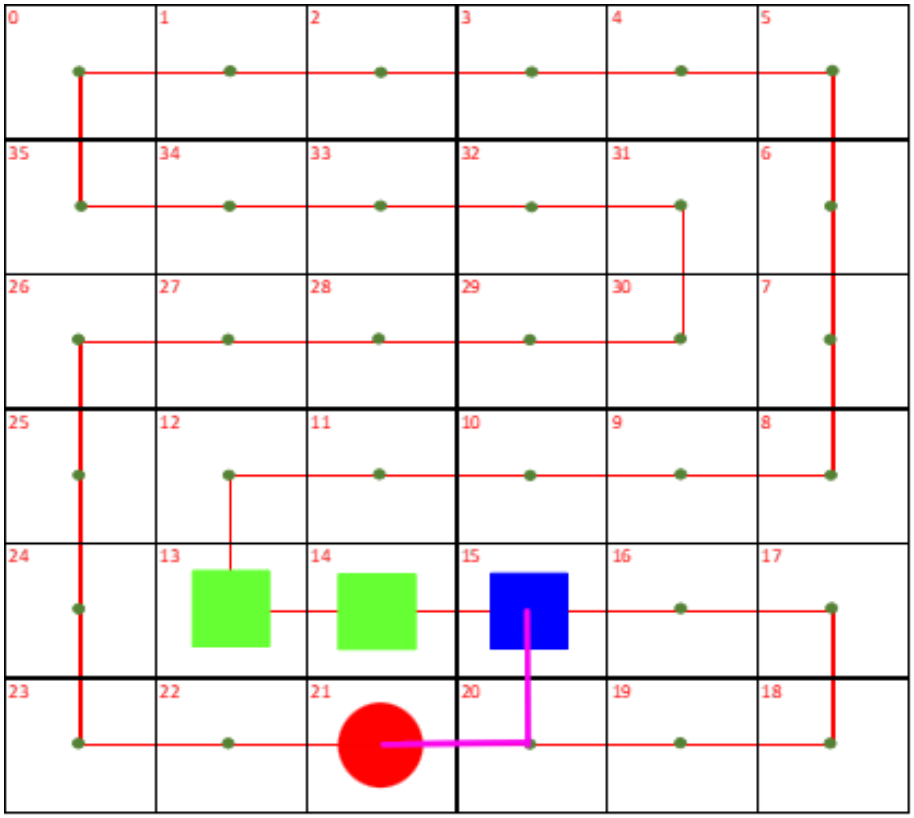
Na příkladu si uveďme, jak přesně bychom museli zkratky vytvářet. Na Obrázku 5.1 vidíme, že had se nachází na políčkách 13, 14 a na políčku 15 je hlava hada. Jablko se nachází na políčku 21. Místo toho, abychom následovali další 4 políčka po kružnici, můžeme udělat zkratku z políčka 15 na políčko 20 a následně se přemístit na políčko 21 a sebrat jablko. Tím ušetříme čtyři tahy.

Zkratky můžeme vytvářet pouze po směru Hamiltonovské kružnice. To znamená, že např. z vrcholu 26 bychom dokázali vytvořit zkratku do vrcholu 35, ale z vrcholu 16 bychom nemohli utvořit zkratku na vrchol 9. Tím docílíme toho, že hlava hada nikdy nepředběhne jeho ocas.

Dále potřebujeme kontrolovat, jestli má had dostatek prostoru pro růst po sebrání jablka, při využití zkratky. Protože pokud využije zkratku a přitom nemá dostatek prostoru na růst dále ve směru kružnice, poroste do sebe a tím pádem sám do sebe narazí a zemře.

V praxi to funguje tak, že pokaždé, když had sebere jablko, najdeme nejkratší cestu k dalšímu jablku a ověříme, jestli zkratka daná touto nejkratší cestou vyhovuje všem námi

dříve stanoveným podmínkám. Pokud ano, had využije zkratku, pokud ne, pokračuje ve sledování kružnice.



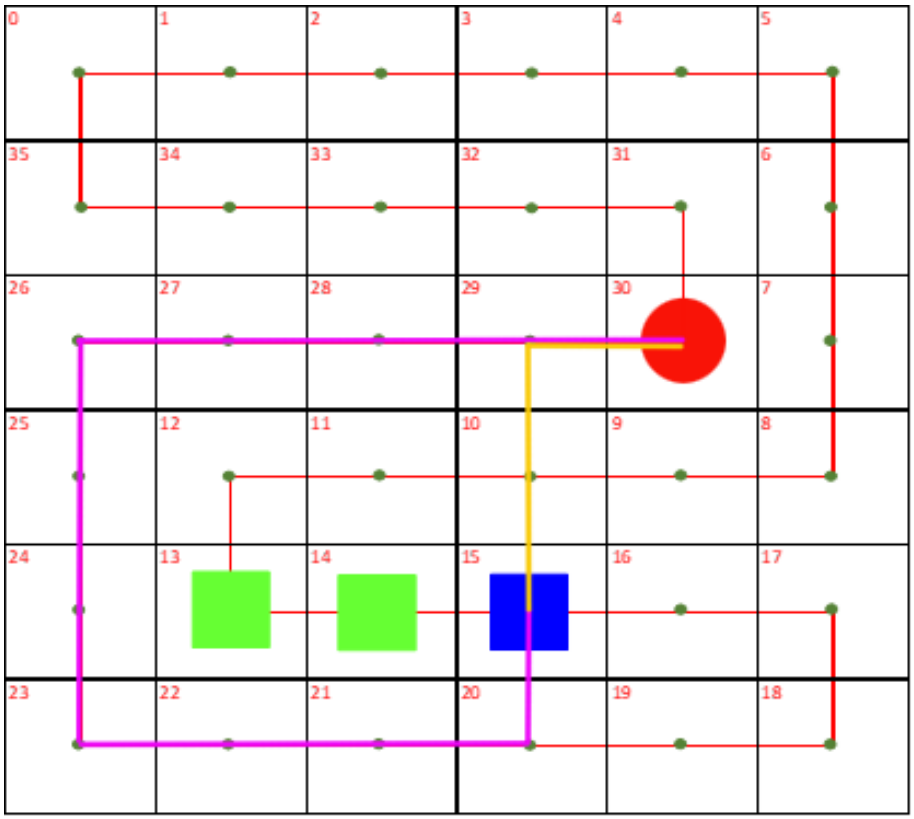
Obr. 5.1: Had následující Hamiltonovskou kružnici, zelené čtverečky - tělo hada, modré čtverečky - hlava hada, červené kolečko - jablko, růžová lomená čára - zkratka, kterou had udělá

5.2 Implementace algoritmu

Vyhovující zkratky budeme generovat pomocí jemně upraveného A* algoritmu. Tedy pokaždé, když had sebere jablko, pomocí upraveného algoritmu A* najdeme nejkratší cestu od hlavy hada k jablku a vyhodnotíme, zda nalezenou zkratku použijeme, nebo ne.

Aby se nalezena cesta dala považovat za zkratku, musí obsahovat pouze takové vrcholy, které jsou vzestupně seřazeny podle pozice v Hamiltonovské kružnici. Například, pokud se jablko nachází na políčku 30 a had se nachází na pozicích 13, 14 a 15 (hlava hada), had nemůže využít zkratku, která by vedla z 15 do 10, pak do 29 a potom by následoval za jablkem do pole 30. Taková cesta nesplňuje výše zmíněné seřazení vrcholů. Nejlepší cesta, kterou může využít, obsahuje zkratku z políčka 15 do 20 a poté následovat Hamiltonovskou

kružnici až k jablku (viz Obrázek 5.2).



Obr. 5.2: Had následující Hamiltonovskou kružnici, zelené čtverečky - tělo hada, modré čtverečky - hlava hada, červené kolečko - jablko, růžová lomená čára - zkratka, kterou had může udělat, žlutá lomená čára - zkratka, kterou had nemůže udělat

Modifikovaná verze algoritmu A* (implementace algoritmu v jazyce Python k nahlédnutí v Ukázce kódu 5.1) povoluje vybrat do cesty pouze ty vrcholy, které jdou vzestupně podle Hamiltonovské kružnice.

```
    def a_star(game_state: GameState, apple_position: tuple[int, int], n_rows: int,
    n_cols: int, hamiltonian_cycle_order):
     open_set = []
     heapq.heappush(open_set, PrioritizedItem(0, game_state))
         for neighbor in current.neighbors(n_rows, n_cols):
             tentative_g_score = g_score[current.head_position] + 1
             neighbor.predecessor = current
             start_head_position = game_to_graph(current.head_position, n_cols)
             end_head_position = game_to_graph(neighbor.head_position, n_cols)

             if neighbor.head_position not in g_score or tentative_g_score <
             g_score[neighbor.head_position]:
                 if hamiltonian_cycle_order[start_head_position] <=
                 hamiltonian_cycle_order[end_head_position] or
                 (hamiltonian_cycle_order[start_head_position] == (n_cols * n_cols) - 1
                 and hamiltonian_cycle_order[end_head_position] == 0):
                     g_score[neighbor.head_position] = tentative_g_score
                     f_score[neighbor.head_position] = tentative_g_score +
                     heuristic(neighbor.head_position,apple_position)
                     neighbor.predecessor = current

                     if neighbor not in [item.state for item in open_set]:
                         heapq.heappush(open_set,
                         PrioritizedItem(f_score[neighbor.head_position], neighbor))

     return None
```

Ukázka kódu 5.1: Modifikovaný A*

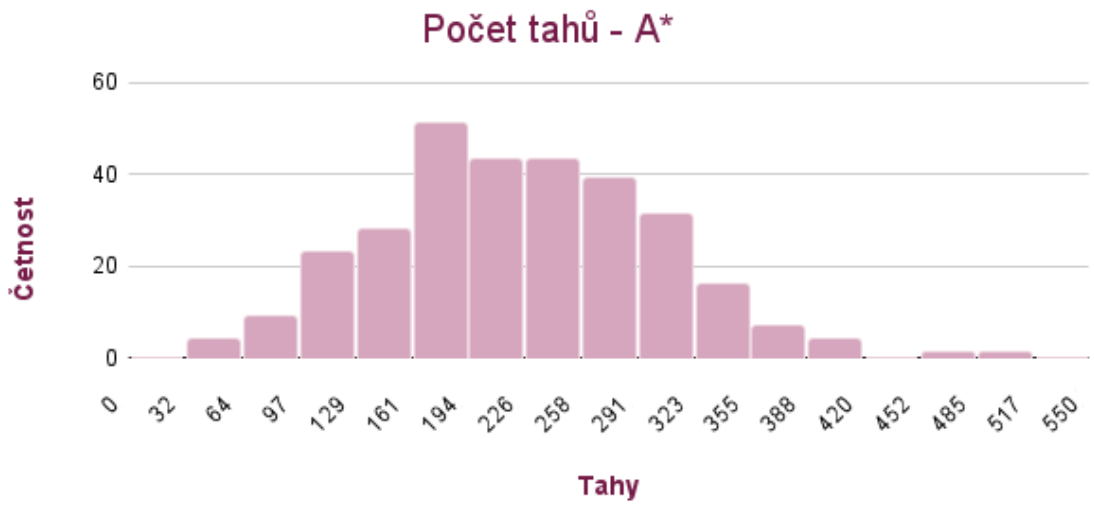
6 Srovnání algoritmů

V této kapitole provedeme srovnání jednotlivých přístupů k řešení hry Snake. Zaměříme se na dvě hlavní kritéria: úspěšnost dokončení hry a efektivitu. Přičemž úspěšností myslíme, jak často algoritmus vede k vítězství - zaplnění celé hrací plochy bez kolize. Efektivitou rozumíme, jak rychle had sbírá jablka a jak plynule se pohybuje bez zbytečných tahů.

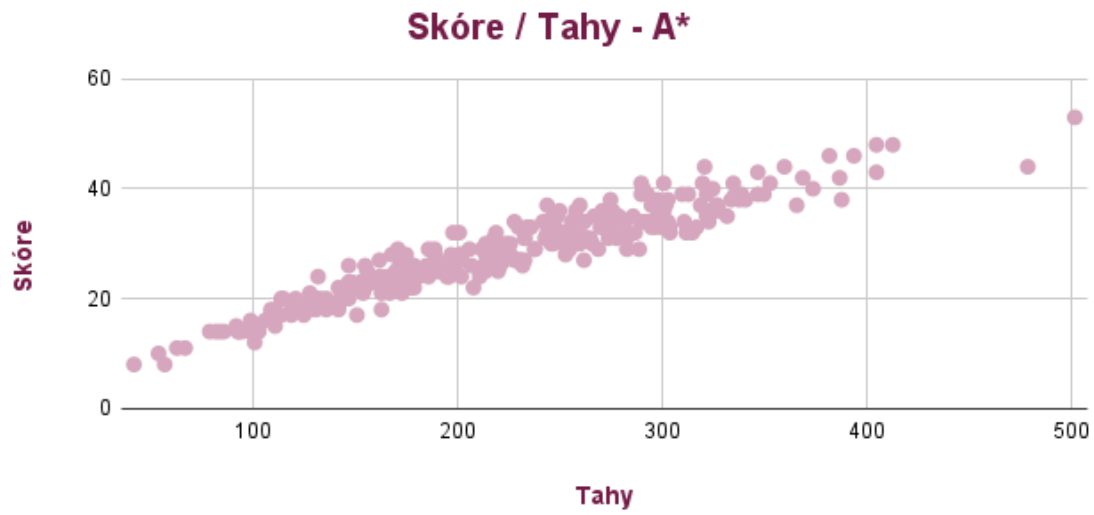
Porovnáme tři přístupy na herním poli 10×10. Maximální dosažitelné skóre je 97. Prvním je A* (viz Kapitola 3), heuristický algoritmus hledající nejkratší cestu od hlavy hada k jablku. Další je Hamiltonovská kružnice (viz Kapitola 4), která představuje předem vygenerovanou cestu, jež zajišťuje, že had nikdy nenarazí sám do sebe. Posledním je algoritmus zkratek (viz Kapitola 5), který kombinuje Hamiltonovskou kružnici s možností vytvářet optimalizované zkratky pomocí algoritmu A*.

6.1 Výsledky A*

Řešení problému s využitím algoritmu A*, který hledá nejkratší cestu od hlavy hada k jablku, nedopadlo nikdy úspěšně. Had z více než 300 pokusů ani jednou nedohrál hru (nezaplnil celou hrací plochu bez kolize). Průměrná délka hry trvala 21 sekund. Had v průměru dosáhl skóre 28 a průměrný počet tahů byl 223. Na Grafu 6.1 můžeme vidět četnosti tahů během různých her. Vidíme, že modus počtů tahů je 161-194. V grafu 6.2 je znázorněna závislost skóre na počtu tahů. Vidíme, že čím více tahů had udělal, tím většího skóre dosáhl.



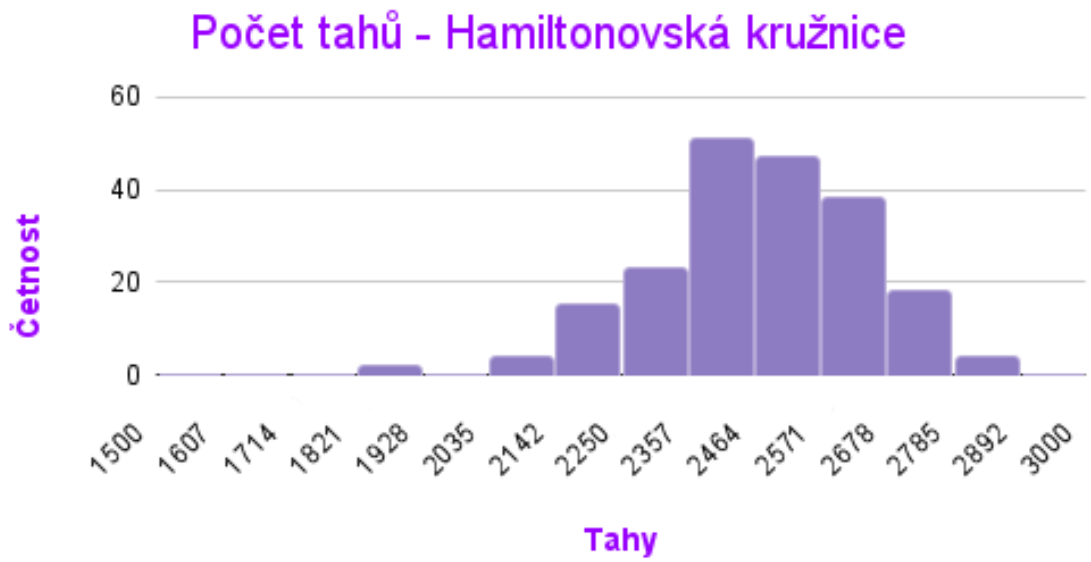
Obr. 6.1: Četnost tahů během různých her - A*



Obr. 6.2: Závislost skóre na počtu tahů - A*

6.2 Hamiltonovská kružnice

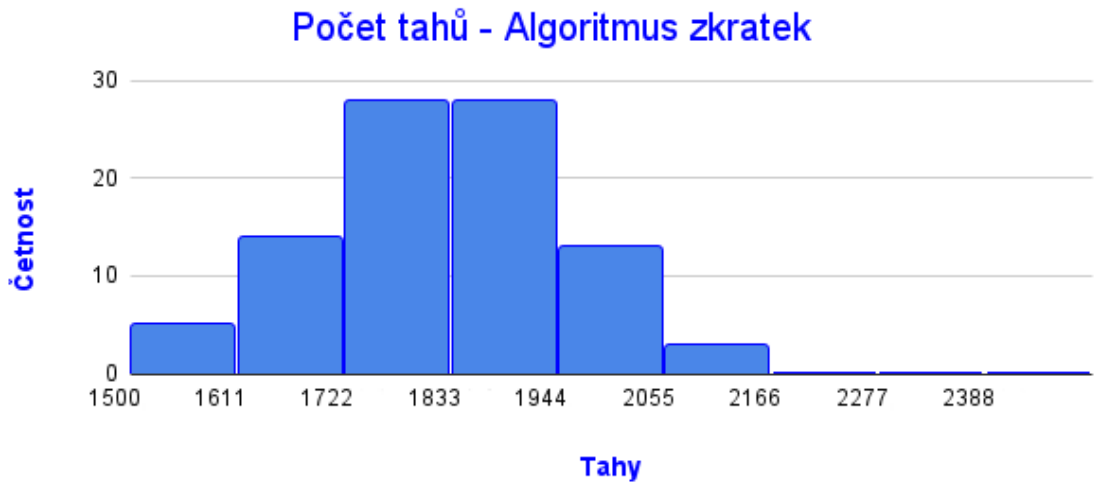
Využití sledování Hamiltonovské kružnice k řešení problému bylo 100% úspěšné. Průměrná délka hry z 200 pokusů byla 4 minuty a 21 sekund a průměrný počet tahů, který závisí na pozicích generovaných jablek, byl 2485. Největší počet tahů, kterého bylo dosaženo, je 2997 a nejmenší počet tahů byl 1921. V Grafu 6.3 je znázorněna četnost tahů během různých her. Je zcela zřejmé, že modus počtů tahů ve hře je 2357-2464.



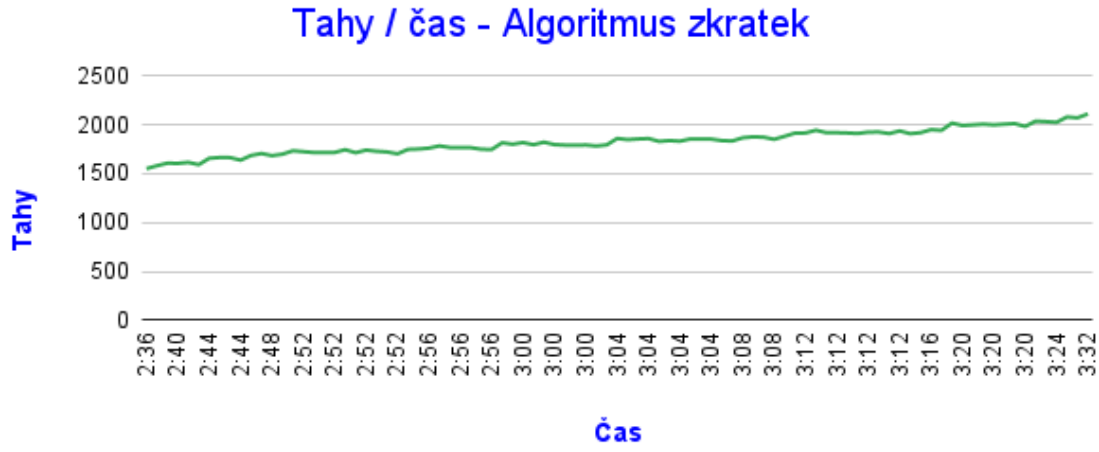
Obr. 6.3: Četnost tahů během různých her - Hamiltonovská kružnice

6.3 Algoritmus zkratek

Algoritmus, který vytváří zkratky v existující Hamiltonovské kružnici, vedl v 72% případech z přibližně 230 pokusů k úspěšnému dokončení hry. V těchto úspěšných pokusech byla průměrná délka hry 3 minuty a 2 sekundy. Průměrný počet tahů, v úspěšných hrách, byl 1825. V Grafu 6.4 je vidět četnost tahů během úspěšných her. Modusy počtů tahů, provedených během hry, jsou 1722-1833 a 1833-1944. Největší počet tahů za jednu hru byl 2112. Tak vysoký počet naznačuje, že had v průběhu hry nevyužil skoro žádné zkratky, protože se počet tahů blíží jednomu z nejmenších počtů tahů, při prostém využití Hamiltonovské kružnice. V Grafu 6.5 je, z pouze úspěšných pokusů, zobrazena závislost počtu tahů na čase.



Obr. 6.4: Četnost tahů během různých her - Algoritmus zkratek



Obr. 6.5: Závislost počtu tahů na čase - Algoritmus zkratek

6.4 Vzájemné porovnání jednotlivých přístupů

Pomocí tabulky 6.1 si porovnáme jednotlivé přístupy. Můžeme vidět, že algoritmus $A*$, kvůli tomu, že hru nikdy nedokončil, má ve všech měřených oblastech velmi odlišné hodnoty, proto budeme srovnávat spíše algoritmus zkratek a Hamiltonovskou kružnici. Hamiltonovská kružnice byla v dokončení hry 100% úspěšná, na rozdíl od algoritmu zkratek, který byl úspěšný pouze ze 72%. Průměrná délka úspěšné hry algoritmu zkratek byla 3 minuty a 2 sekundy, což je přibližně o 30% lepší čas, než u prostého sledování Hamiltonovské kružnice. Průměrný počet tahů, v úspěšných hrách algoritmu zkratek, byl 1825 a to je skoro

o 100 tahů méně, než nejmenší počet tahů při využití pouze Hamiltonovské kružnice. Modusy tahů v úspěšných hrách algoritmu zkratek jsou výrazně menší než modus tahů na hru u Hamiltonovské kružnice.

Algoritmus	Úspěšnost v %	Průměrný čas hry	Průměrný počet tahů na hru	Modus tahů na hru
A*	0	21 s	223	194
Hamiltonovská kružnice	100	4 min 21 s	2485	2357-2464
Algoritmus zkratek	72	3 min 2 s	1825	1722-1833, 1833-1944

Tabulka 6.1: Porovnání jednotlivých přístupů k řešení hry Snake. Porovnáváme úspěšnost dokončení hry, průměrný čas pro dokončení hry, průměrný a nejčetnější počet tahů na hru pro algoritmy A*, Hamiltonovskou kružnici a algoritmus zkratek.

Tato zjištění ukazují, že i když Hamiltonovská kružnice garantuje 100% úspěšnost, algoritmus zkratek je efektivnější, pokud jde o rychlost dokončení hry a počet tahů. Algoritmus zkratek poskytuje rychlejší a plynulejší hru s nižším počtem tahů, což jej činí nejefektivnějším přístupem mezi těmito třemi algoritmy pro řešení hry Snake.

7 Závěr

V této práci jsem analyzoval a porovnal tři různé algoritmy pro řešení hry Snake: A*, Hamiltonovskou kružnici a algoritmus zkratek. Hodnotil jsem je z hlediska úspěšnosti dokončení hry a efektivity sbírání jablek. Při testování jsem zjistil, že samotný algoritmus A* nebyl nikdy úspěšný, protože had se dříve či později dostal do situace, kdy narazil do sebe. Ačkoliv A* efektivně hledal nejkratší cestu k jablku, nebyl schopen zajistit dlouhodobě bezpečný pohyb hada.

Hamiltonovská kružnice se ukázala jako stoprocentně spolehlivá strategie, která vždy vedla k úspěšnému dokončení hry. Jejím hlavním nedostatkem však byla nízká efektivita – had často prováděl zbytečně dlouhé tahy, což značně prodlužovalo dobu trvání hry.

Nejlepším kompromisem se ukázal být algoritmus zkratek, který kombinuje Hamiltonovskou kružnici s možností vytvářet optimalizované cesty pomocí modifikovaného algoritmu A*. Tento přístup vedl k dokončení hry v přibližně 72% případech a zároveň výrazně zkrátil počet tahů i celkovou dobu hry v porovnání s čistým následováním Hamiltonovské kružnice.

Při implementaci jsem narazil na několik zásadních problémů. Nejnáročnější bylo nalezení Hamiltonovské kružnice pro dané herní pole. Tento problém patří mezi NP-úplné problémy, což znamená, že neexistuje efektivní algoritmus pro jeho řešení v obecném případě. Musel jsem proto využít speciální postupy pro nalezení kružnice. Dalším problémem byla efektivní detekce možných zkratek, která musela respektovat pořadí vrcholů v Hamiltonovské kružnici.

Nakonec se mi podařilo vytvořit funkční a relativně efektivní řešení, přesto stále vidím prostor pro další zlepšení. Možným rozvojem projektu by bylo například dynamické upravování Hamiltonovské kružnice podle aktuální situace na herním poli nebo využití pokročilejších heuristik pro rozhodování o použití zkratek.

Celkově hodnotím projekt jako úspěšný, protože se mi podařilo implementovat a otestovat různé strategie řízení hada, porovnat jejich výkonnost a identifikovat nejefektivnější přístup.

Zároveň jsem získal cenné zkušenosti s algoritmy pro hledání cest a jejich aplikací na problém s dynamickým prostředím.

Seznam ukázek obrázků

Seznam ukázek kódu

Seznam zdrojů

[1] Aditya Sharma. *What is Dijkstra's Algorithm? | Introduction to Dijkstra's Shortest Path Algorithm*. Accessed: Mar. 29, 2025. [Online]. Available: https://www.geeksforgeeks.org/introduction-to-dijkstras-shortest-path-algorithm/. Břez. 2025.

[2] Neuvedeno. *Hamiltonian Cycle*. Přístupné online: 21. března 2025. [Online]. Dostupné z: https://www.geeksforgeeks.org/hamiltonian-cycle/. Břez. 2025.

[3] Neuvedeno. *Math Tutor - Functions - Theory - Elementary Functions*. Přístupné online: 29. března 2025. [Online]. Dostupné z: https://math.fel.cvut.cz/mt/txtb/4/txc3ba4b.htm. Břez. 2025.

[4] Soumyadeep Debnath. *Big O Notation Tutorial - A Guide to Big O Analysis*. Přístupné online: 29. března 2025. [Online]. Dostupné z: https://www.geeksforgeeks.org/analysis-algorithms-big-o-analysis/. Břez. 2025.

[5] Erik Demaine. *16. Complexity: P, NP, NP-completeness, Reductions*. Přístupné online: 21. března 2025. [Online]. Dostupné z: https://www.youtube.com/watch?v=eHZifpgyH_4. Břez. 2016.

[6] John Tapsell. *snake*. Přístupné online: 22. března 2025. [Online]. Dostupné z: https://johnflux.com/tag/snake/. Břez. 2025.

[7] D. Graafsma. *Playing Snake on a Graph*. Přístupné online: 29. března 2025. [Online]. Dostupné z: https://essay.utwente.nl/102968/7/Graafsma_MA_EEMCS.pdf. Břez. 2025.

[8] Brian Haidet. *How to Win Snake: The UNKILLABLE Snake AI*. Přístupné online: 24. března 2025. [Online]. Dostupné z: https://www.youtube.com/watch?v=TOpBcfbAgPg. Břez. 2020.