Vývojová dokumentace

Zadání

Vytvořit piškvorky se základním grafickým vzhledem v jazyce python za použití modulu pygame. Hlavní část je vytvoření protivníka, který bude sám vyhledávat nejlepší tahy.

Grafická stránka

Protože jsem vytvářel hru, musel jsem vytvořit i velice základní grafické rozhraní. K tomu jsem využil funkce, které nabízí modul pygame. Mřížku jsem vykreslil pomocí pygame.draw.rect. Nejdříve jsem nechal vykreslit černé pozadí a následně jsem s určitými rozestupy vykresloval malé čtverečky jako políčka. Toto řešení má výhodu, že jednoduchým změněním proměnné „rowCount“ se mi zvětší celé herní pole. To bych nemohl, kdybych si pole nakreslil v nějakém grafickém programu a poté ho jen zobrazil. Na druhou stranu by to bylo kódově jednodušší.

Tímto způsobem jsem si ale vytvořil křížky a kolečka. V malování jsem udělal obrázky a ty se mi zobrazují na určeném místě.

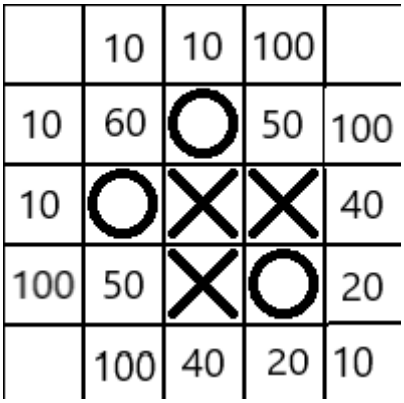
Díky funkci is_valid() se mi po kliknutí mimo hrací pole hra nerozbije. Zároveň při kliknutí na již obsazené pole, se toto kliknutí nepočítá.

Posledním grafickým prvkem je ukázání vítěze. To je opět dělané jako obrázek zobrazený přímo doprostřed herního pole. Tuto část bych nejspíše v další hře udělal jinak, protože zbytečně zakrývá dohrané hrací pole.

Další úpravy do budoucna by mohlo být například skóre, kdy každý hráč dostane za vítězství bod a může se hrát tímto způsobem. Nebo by se dal program vylepšit za pomocí zvýraznění posledního tahu. Ten se na velkém zaplněném hracím poli občas těžko hledá. To jsou však potenciální plány do budoucna. Se současnou grafickou stránkou jsem spokojený. Zároveň si myslím, že je v tomto případě mnohem důležitější programovací část než ta grafická.

Algoritmus

Hlavní část programování bylo vytvořit algoritmus, který dokáže nalézt optimální tah. K tomu jsem využil algoritmus, který zkontroluje okolí již obsazených pozic. Ty následně ohodnotí a na pozici s nejvyšším hodnocením vloží svůj „kamen“. Přednost dává hodnocení vedle svých kamenů než vedle kamenů protivníka. Příklad ohodnocení je možné vidět na obrázku číslo 1.



Obrázek 1 - Příklad ohodnocení

Ke správnému chodu tohoto algoritmu potřebuji třídu „Board“, ve které mám uložené všechny pozice. Každá z pozic k sobě ještě ví jaký kámen je na ní položený a její hodnocení. Další věc, kterou potřebuji je list „newTick“, do kterého se ukládají pozice zahraných tahů.

List „newTick“ není nutný, zaručí však, že se nemusí prohledávat všechny pozice, ale jen okolí míst, kde víme, že již kameny jsou. Na takovýchto místech bude totiž s největší pravděpodobností nejlepší pozice pro další kámen.

Po zavolání tohoto algoritmu se vezme první políčko z listu „newTick“ a zkontrolují se jeho sousedé. O to se stará funkce „possibleTurns()“. Pokračovat se dá jen na ty sousedy, na kterých není umístěn žádný kámen. Pokud v okolí není žádné volné místo, odebereme toto políčko z listu „newTick“. Díky tomu se příště nebude program zbytečně zdržovat kontrolou polí, ze kterých nikam nemůže. Pokud se zjistí, že na nějakém poli místo je, zavolá se funkce „turn()“. Ta kontroluje, zda už toto pole nemá nějaké score, protože to by znamenalo, že už se kontrolovalo a je tedy zbytečné ho kontrolovat znovu. Tyto dvě funkce tedy hledají místa, na která se dá vložit kámen.

V dalším kroku se zavolá funkce toWhichSide(), která se stará o to, aby se prozkoumali všechny možné směry. Tyto směry se dají omezit na čtyři: Vertikální, horizontální, a dva diagonální. Kdy naše políčko je vždy uprostřed. Po vybrání některého ze směrů se volá fukce „choosingPlace()“. Ta využívá funkci „whatIsThere()“, k tomu, aby zjistila, jaké kameny jsou umístěny v určených směrech. Tyto dvě funkce se tedy starají o zjištění, co se kolem naší potenciální pozice nachází.

Následující částí je ohodnocovací část. Tedy část, která dá každému políčku skóre. To se děje ve funkci „scoreOfTile()“. Ta vždy využije počet ticků a prázdných míst v určeném směru a podle toho dá našemu políčku nějaké hodnocení. Nejvyšší jsem

nastavil na 200 000. Toto je výherní políčko. Od něj se následně odvozuje i game_over. S hodnocením zároveň pomáhá funkce toWhichSide(). Ta dostane vždy čtyři skóre, každé pro jeden směr, a z nich vybere to nejvyšší. Pokud se však opakují skóre vyšší než 40000, pak se tato skóre sčítají. To se následně zapíše do třídy Board buď k rankX, pokud se řeší Xka, nebo rankO, pokud se řeší kroužky.

A jako poslední je kód, který má za úkol najít to nejlepší pole podle skóre. Za to se zaručuje funkce bestTurn(). Ta prohledá všechna ohodnocen místa. Tato jsem si totiž po celou dobu ukládal do listu „positions“. Díky tomu nemusí funkce bestTurn() prohledávat úplně celé pole, ale stačí porovnat jen ta místa, která se ohodnotila. Poté, co to najde nejlepší místo, zkontroluje, jestli tento tah náhodou neukončuje hru, tedy jestli se skóre nerovná 200 000 a pak tuto pozici pošle zpět do funkce main(), která to vrátí do hlavního programu.

Speciální případ nastane, když nejlepší možný tah je vytvořit trojici. Tedy že skóre je 40 000. V tom případě se zjistí, zdali není možné v nejbližším tazích vytvořit jedním kamenem dvě trojice. To totiž znamená jasnou výhru. O tuto část se stará kód v souboru „turns.py“.

Plánování tahu dopředu

Zavoláním funkce ifThreeIsPossible() toto hledání začíná. Po celou dobu chodu kódu se do listu twoInARow ukládali všechny možné varianty, jak dojít ke třem stejným kamenům v jeden tah. To musí totiž protihráč blokovat, jinak prohraje. Pokud mu ale uděláme dvě takové trojice v jeden tah, pak už má prohru téměř jistou.

Proto jsem se rozhodl, že tento tah bude jediný, který budu plánovat na alespoň jedno kolo dopředu. Přicházím tím sice nejspíše o nějaké možnosti, jak vyhrát, ale ušetřím velké množství času.

Hledání vhodného tahu funguje tak, že si zjistím, co se nachází na okolních pozicích. Pokud se na tato místa dá vložit kámen, nebo je na nich mnou preferovaný znak, pak pokračuji v hledání za nimi. Zároveň si ale musím uvědomit, že pokud chci udělat dobrou trojici, pak potřebuji, aby měla volné místo na obě strany. Proto tedy, když bude zablokovaný pohyb do prava, pak nemá cenu hledat vlevo. Tímto dokážu oříznout část zbytečného hledání. Následně zjistím, jaké proměnné se v mnou prohledávaných řadách nachází. Pokud mnou nepreferované zaškrtnutí, pak tento řádek automaticky zahazuji, protože by se dal jen s těží rozšiřovat na 5.

To samé platí, pokud se v řádku nenachází ani jeden můj kamen. Poté by trvalo příliš dlouho vystavět tři vedle sebe. Proto jediná možnost, se kterou pokračuji je, když se v řádku nachází alespoň jeden můj kámen a žádný cizí. Poté si vyberu lepší pozici na umístění vedle něj a tam umístím kámen. Zároveň si uložím, že v příštím tahu chci vytvořit dvě trojice.

Mohl jsem si plánovat více tahů dopředu. Rozhodl jsem se ale, že kvůli vyšší časové náročností se mi to nevyplatí. Raději budu mít program malinko hloupější, než moc pomalý.

Alternativní řešení

Další způsob, který se dá uplatnit je vytváření všech kroků dopředu pro úplně všechny tahy. To je ale příliš náročné jak na paměť, tak na čas. Proto přišlo jako první omezení to množství tahů. Ve většině případů stačí 1-3 tahy dopředu. Stále je ale tento přístup příliš náročný. Proto jsem se rozhodl prohledávání omezit pouze na místa, kde je možné vytvořit „určitou výhru“. Tedy ta místa, kde se dá teoreticky umístit kámen pro dvě trojice nebo čtveřice najednou. Tímto jsem dokázal čas i potřebné místo výrazně zmenšit. Přicházíme sice o neporazitelnost, některé skvělé tahy se tím dají přehlédnout, ale pokud by se každé místo vybíralo 10 minut, nikoho by hraní nebavilo.

Algoritmus, který jsem i dokonce naprogramoval, bylo prohledávání okolí pouze kolem posledních dvou umístěných kamenů. To dokázalo velice zkrátit čas i potřebné místo, ale přišlo se až o příliš mnoho možností. V nějaké budoucí hře by se ale tato možnost dala využít jako „lehká verze“ protivníka, protože většina tahů se opravdu děje jen kolem několika posledních kamenů.

Program

Největší část programu je uložena ve třídě „Board“. Toto jsem využil hlavně kvůli snížení množství různých listů a kvůli lehkosti zapisování. Stačí jednoduše zavolat odkudkoliv. Zároveň se zde jednoduše upravuje jakákoliv proměnná. Na to jsem si vytvořil funkce „update“. Z důvodu lepší orientace jsem program rozdělil do tří souborů. „main.py“ řeší vzhled a hlavní fungování. Soubor „comp.py“ (zkrácené slovo computer) se stará o chod protihráče. „turn.py“ hledá několik tahů dopředu. Díky tomu je program přehlednější.

Dá se říct, že všechny ostatní listy, které používám jsou takové fronty. Čtu v nich popořadě. Žádnou proměnnou ale neodebírám, pokud se neřekne jinak. Není to však nutné. U všech listů bych mohl číst klidně na přeskáčku, tato možnost je ale nejlehčí.

Celý program, až na komentáře, je psán anglicky. A to z důvodu, že v „reálném světě“ mohu takový program poslat klidně spolupracovníkovi do Jižní Koreji a on bude vědět, co se zde děje. Komentáře jsem se ale nakonec rozhodl napsat česky, protože tento program bude v tuto chvíli číst pouze omezené množství lidí a nemám v plánu ho šířit do zahraničí.

Závěr

Tento program by se dal ještě zrychlit jako i zlepšit po té „inteligenční“ stránce programu. Po každém tahu se například vymaže veškeré hodnocení všech pozic. To samozřejmě není nejlepší způsob, většina hodnocení zůstane stejná. Rozhodl jsem se ale věnovat čas zrychlování v jiných místech programu a vylepšování hledání tahu dopředu.

Do budoucna by se dala dále zlepšit vizuální stránka, přidání možností výběru velikosti pole a toho kolikátý kdo bude hrát a proti komu. Na tyto věci je program připraven, ale čas si řekl jinak.

Z výsledku mám ale dobrý pocit. Do tohoto úkolu jsem šel s tím, že v piškvorkách prohraji i na poli 3x3. Proto pro mě byla výzva vymyslet algoritmus, který mnohonásobně překoná svého stvořitele. Není to dokonalé, proti ostatním „robotům“ vyhraje tak 2 hry z deseti, ale normální hráč si zahraje dobře a většinou i prohraje. Tento úkol měl tedy dvě výhody. Naučil jsem se programovat a hrát piškvorky.