Končno poročilo projekta: Igra 2048

Maj Gaberšček

Ljubljana, april 2022

1 Uvod

Pri predmetu Matematika z računalnikom si je vsak študent izbral svoj projekt med nalogami, ki so vključevale projekte raznih industrijskih panog, projekte iz optimizacije, teorije iger,... Sam sem si izbral projekt z naslovom *Games 3: 2048*. V sklopu projekta sem si zastavil nalogo, da uspešno zasnujem uporabniški vmesnik za igranje igre 2048. Ta bo uporabniku omogočal, da sam odigra igro, ali pa izbere računalniški algoritem, ki nato odigra igro in jo skuša premagati. Mogoča je tudi kombinacija: igralec odigra igro do neke poteze, potem pa jo naprej skuša rešiti računalnik, ali obratno. Za implementacijo sem si izbral programski jezik `Java`, ki s knjižnico `Swing` omogoča relativno preprosto implementacijo uporabniškega vmesnika. Projekt sem sproti objavljal tudi na repozitorij na `Githubu`.

2 Predstavitev igre 2048

2.1 Predstavitev

2048 je enoigralska video igra, ki jo je izumil italijanski razvijalec Gabriele Cirulli leta 2014 in jo objavil na `Github`. Cilj igre je skupaj sestavljati različne številke na mreži in doseči število 2048. Mreža je praviloma velikosti 4x4, čeprav se v različnih variantah igre pojavlja tudi v drugih velikostih (na primer 3x3 ali 5x5) [1]. V teoriji iger je sicer dokazano, da je problem igre 2048 NP-poln.[2]

2.2 Pravila

Uporabnik lahko, ko je na vrsti, naredi potezo, pri čemer ima ponavadi na voljo 4 različne: premik gor, premik dol, premik levo in premik desno. Ob premiku v želeno smer se vsako število premakne najdlje možno v tisto smer, dokler ni zaustavljeno ali s koncem mreže, ali z drugo številko. Pri tem se, če se zaletita dve enaki števili, združita v novo število, ki je vsota obeh (dvakratnik). Upoštevati moramo, da se, v kolikor se zaletijo tri enaka števila, v dvakratnih združita samo tisti dve, ki sta najdlje v smeri premika. Poteze, ki mreže ne spremeni (torej ostane enaka kot pred premikom), ne smemo odigrati.

Po vsakem premiku se na naključno prazno mesto na mreži pojavi novo število, ki je 2 z verjetnostjo 90% in 4 z verjetnostjo 10%.

Igra je izgubljena, ko uporabnik nima več možnih potez, torej nobena poteza ne spremeni mreže. Če uporabnik doseže število 2048, je igro premagal. Večina aplikacij sicer uporabniku omogoča tudi igranje naprej po zmagi. Z ustrezno strategijo je namreč igro precej preprosto premagati.

2.3 Točkovanje

Igra ponuja tudi točkovanje. Po zmagi je namreč cilj uporabnikov to, da dosežejo čim več točk.

Točke se zbirajo tako, da se vsakič, ko se dve števili združita v njuno vsoto, le-ta prišteje točkam.

2.4 Strategija človeka

Ljudje, ki igro igrajo, ponavadi uporabijo strategijo, da največje število na mreži hranijo v določenem kotu, zraven nje pa ostala velika števila, pri čemer je cilj predvsem v tem, da največje število ne skoči iz kota.

Potrebno je tudi poudariti, da igre seveda ni mogoče igrati v nedogled. Največje število, ki ga lahko z idealno igro in precej sreče (samo teoretično) dosežemo, je 131072.

3 Načrt dela

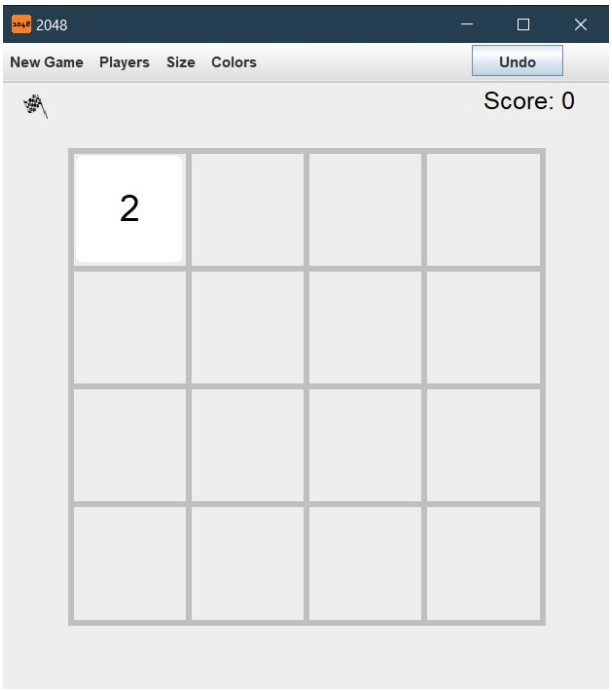
Najprej sem v programskem jeziku `Java` ustvaril razred `Game.java`, ki predstavlja igro. Na razredu sem implementiral metode, ki služijo premikom, pojavitvam novih števil,...

Nato sem s pomočjo knjižnice `Swing` definiral dva nova razreda. Razred `Panel.java` je nekakšno platno, ki služi za risanje igre in podpira interakcijo uporabnika s tipkami (premiki na igri) Razred `Frame.java` je okno, ki vsebuje platno. Okno vsebuje tudi menijsko vrstico, s pomočjo katere izbiramo med nastavitvami igre in igralci.

Zadnji korak je vključeval zasnovo algoritmov, ki (uspešno) rešujejo igro in nato implementacija delovanja teh algoritmov na igri. Algoritmi so predstavljeni kot metode na razredu `Game.java`, pri čemer klic metode odigra potezo, ki jo posamezen algoritem izbere. Algoritmi so različno uspešni, predvsem pa velja pravilo, da je uspešnost algoritmov ponavadi obratno sorazmerna s časom na potezo.

4 Uporabniški vmesnik

Na Sliki 1 je posnetek zaslona uporabniškega vmesnika aplikacije. Kot lahko vidimo, se čez večino okna nariše mreža (v primeru klasične igre je velikosti



Slika 1: Posnetek zaslona uporabniškega vmesnika, ko aplikacijo zaženemo

4x4). Vsakič, ko aplikacijo zaženemo, se privzeto pojavi nova igra velikosti 4x4, ki jo lahko igramo. Igro igramo s pomočjo tipk *w, a, s, d*.

Zgoraj desno imamo napis *Score*, ki nam sporoča trenutno število točk, ki smo jih dosegli. Zgoraj levo je ikonica, ki nam sporoča, kateri način igre trenutno igramo. V aplikacijo sem sprogramiral dva načina igranja: način *Classic*, ki pomeni, da se igra konča, ko dosežemo število 2048, ter način *Endless*, ki omogoča igranje, dokler ne izgubimo. Način igranja izberemo v menijski vrstici, pod zavihkom *New Game*.

V menijski vrstici lahko pod zavihkom *Players* izbiramo igralca. Izberemo lahko igralca *Player* in potem igro igramo sami, ali pa izberemo igralca *Computer*. V slednjem primeru bo izbrani računalniški algoritem (ki ga izberemo pod zavihkom *Computer Algorithm*) poskušal odigrati igro. Če želimo med igro spremeniti igralca, samo izberemo drugega igralca v tem podmeniju. Na primer, aplikacija omogoča, da računalnik odigra do neke poteze in potem naprej odigramo mi ali obratno.

V zavihku *Size* v menijski vrstici izbiramo velikost mreže igre. Pri tem je potrebno paziti, saj se vsaka sprememba odraža v novi igri (staro igro izgubimo). Implemenirane so velikosti igre 3x3, 4x4, 5x5, 6x6 in 8x8.

Zavihek *Colors* nam omogoča, da si izberemo barvno shemo, ki spremeni barve. Barvne sheme lahko menjamo med igranjem, brez da to vpliva na igro.

Desno od menijske vrstice je še gumb z napisom *Undo*, ki služi razveljavitvi

poteze. Medtem, ko igra računalnik, gumb ne deluje. Prav tako med igranjem računalnika tudi ne delujejo tipke, ki odigravajo človekove poteze (w, a, s in d). Zavedam se, da gumb *Undo* lahko uporabljamo za goljufanje igre (manipuliramo kje na mreži se bo pojavila nova številka). Vseeno je tak gumb koristen, če na primer po pomoti odigramo potezo, ki je nismo želeli odigrati.

Okno z igro je spremenjive velikosti, vendar igra 4x4 najlepše izgleda v privzeti velikosti. Če okno preveč pomanjšamo, bodo številke postale prevelike za prikaz na mreži in se jih ne bo videlo.

5 Računalnikovi algoritmi

Implementiral sem štiri različne računalnikove algoritme za reševanje igre, ki so med seboj različno uspešni.

5.1 Algoritem `Random moves`

Kot nam že ime samo po sebi pove, ta algoritem uporablja naključne poteze za reševanje igre. Algoritem sicer ni pretirano uspešen, je pa podlaga za bolj uspešna in kompleksna algoritma, opisana v naslednjih dveh podpoglavjih.

5.2 Algoritem `Simulator(k)`

Ta algoritem je v teoriji iger znan kot **Pure Monte Carlo game search**. Algoritem sprejme naravno število k. Nato določi vse možne poteze, ki jih v dani poziciji na mreži lahko odigra. Za vsako izmed teh nato v ozadju odigra k naključnih iger, kjer najprej odigra izbrano potezo, nato pa dokonča igro z algoritmom `Random moves`, ki je opisan v prejšnjem podpoglavju, dokler ne izgubi. Nato primerja igre različnih potez in izbere tisto potezo, katere igre (simulacije) so v povprečju dosegle največje število točk, preden so izgubile.

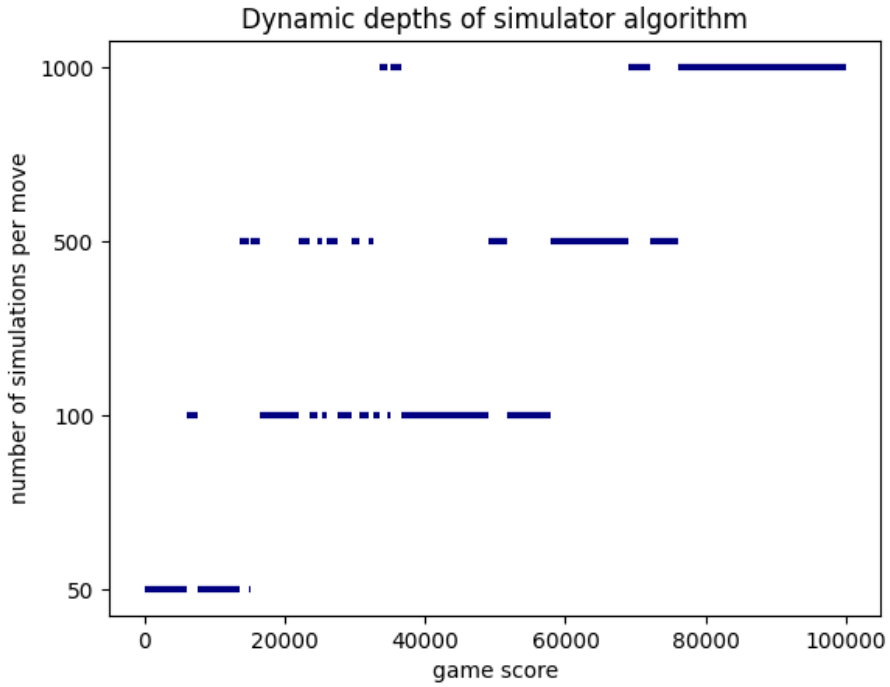
Na primer, v dani poziciji sta možni potezi *Premik gor* in *Premik desno*. Recimo, da uporabimo število $k = 10$. Algoritem v ozadju ustvari 10 kopij igre in na vsaki od njih najprej odigra potezo *Premik gor*. Nato vse igre dokonča z algoritmom *Random moves*. Algoritem v neko spremenjivko shrani povprečno število točk, ki jih je dosegel pri teh 10 igrah. Nato postopek ponovi za potezo *Premik desno*. Ustvari 10 kopij originalne igre in na njih odigra *Premik desno*, ter igre dokonča z naključnimi potezami, dokler ne izgubi vsake. Spet si v drugo spremenljivko shrani povprečno število točk na teh 10 igrah. Nato primerja povprečno število točk za potezo *Premik gor* in *Premik desno*. Odigra tisto, ki je v povprečju dosegla več točk.

5.3 Algoritem `Dynamic simulator`

`Dynamic simulator` je algoritem, ki je izboljšava algoritma `Simulator`. Ta ima namreč problem, da za nezahtevne pozicije porabi preveč časa, da izračuna potezo. Največ težav imajo vsi algoritmi tik pred tem, da izgradijo novo največjo številko, saj je takrat na mreži največ zasedenih polj. Na primer, preden

algoritem zgradi število 2048, mora imeti na mreži že števila 1024, 512, 256, 128, 64. Takoj po tem ko število 2048 zgradi, ima vsa polja na mreži prosta, razen tistega z 2048. Torej bo precej večja verjetnost, da izgubimo tik pred izgradnjo novega najvišjega števila, kot takoj po tem.

Zato je `Dynamic simulator` sestavljen tako, da glede na stanje na mreži ustrezno prilagaja število k algoritma `Simulator(k)`. Če je pozicija bolj zahtevna, uporabimo večji k (in je poteza bolj časovno zahtevna). Če pa je pozicija lažja, lahko uporabimo manjši k. Algoritem je tako časovno manj zahteven od navadnega `Simulatorja`. Prikaz odvisnosti k-ja od števila točk, doseženega na igri je prikazana na grafu na Sliki 2. Uspešnost tega algorima sem analiziral v poglavju *6. Analiza uspešnosti*.



Slika 2: Število simulacij na potezo v odvisnosti od točk za `Dynamic simulator`

5.4 Algoritem `Empty spaces`

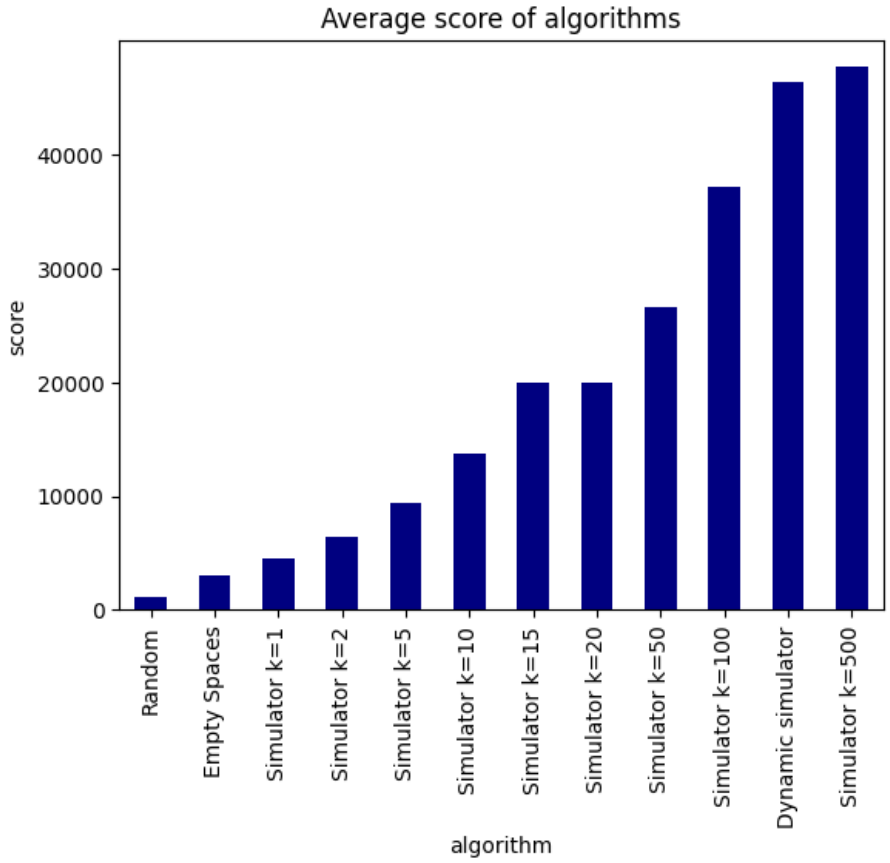
Ta algoritem za vsako potezo pogleda število praznih mest na mreži po njej. Izbere potezo, ki ima najmanjše število praznih polj. Če ima več potez enako število praznih mest, izbere po preferenčnem seznamu: *Premik gor, Premik desno, Premik levo, Premik dol.* Algoritem sicer dobro posnema človekovo strategijo, vendar ni pretirano uspešen pri igranju. Odpove namreč v situacijah, ko je treba predvideti več kot eno potezo v naprej.

6 Analiza uspešnosti

Algoritmi so različno uspešni v reševanju igre in različno hitri. Za vsak algoritem in za $k \in \{1, 2, 5, 10, 15, 20, 50, 100, 500\}$ za algoritem `Simulator(k)` sem igro pognal 100-krat in analiziral uspešnost posamezega algoritma. Najprej je treba poudariti, da je hitrost `Simulatorja` linearno odvisna od števila simuliranih iger k. Zato hitrosti ne bom primerjal med njimi.

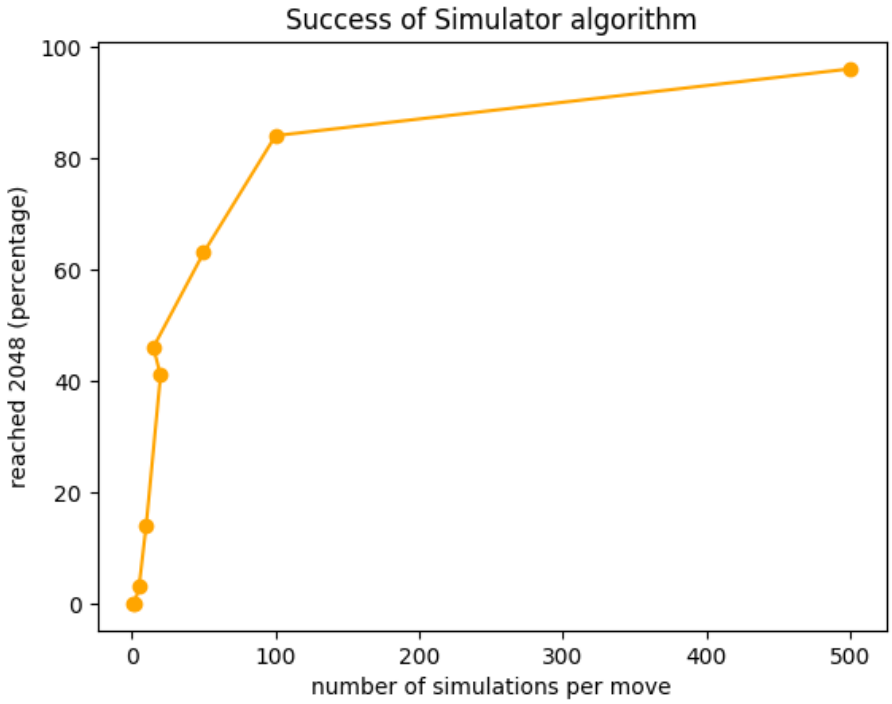
Najhitrejši je algoritem naključnih potez, `Random moves`, sledi pa mu `Empty spaces`. `Simulator` je hitrejši od `Dynamic simulatorja` za manjše k-je, za večje pa je kar precej počasnejši.

Uspešnost algoritmov glede na povprečno največje dosežene točke, je prikazan na Sliki 3. Čeprav je `Dynamic simulator` malce slabši od `Simulatorja(500)`, pa je več kot štirikrat hitrejši. V povprečju `Simulator(500)` rabi za dosego števila 2048 približno 210 sekund, pri čemer `Dynamic simulator` igro v povprečju premaga v manj kot 40. sekundah.



Slika 3: Povprečno število točk, glede na posamezen algoritem

Najpomembnejši kazalec uspešnosti algoritmov je, glede na naravo naloge, uspešnost pri premagovanju igre. Uspešnost `Simulatorja` prikazuje graf na Sliki 4. Algoritma `Random moves` in `Empty Spaces` v 100 poskusih igre nista uspela premagati. Algoritem `Dynamic Simulator` je igro premagal 92-krat od 100 poskusov.

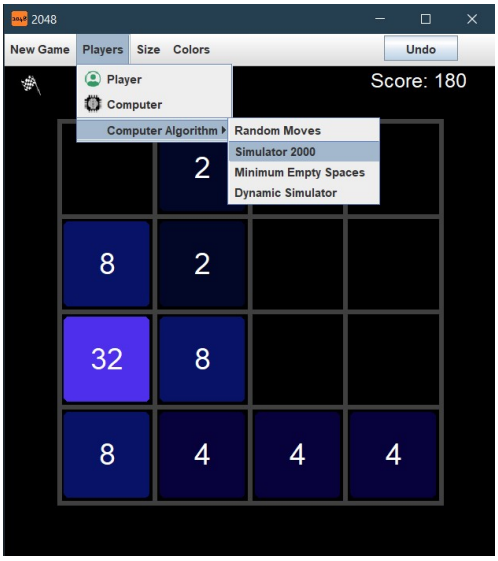


Slika 4: Odvisnost števila simulacij od uspešnosti za `Simulator`

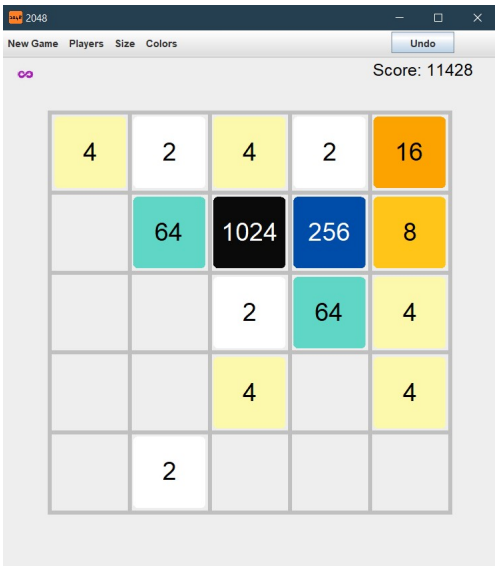
Nekako se zdi, da uspešnost logaritemsko narašča s številom simulacij. Za boljši pregled odvisnosti bi bilo potrebno testirati tudi vrednosti med $k = 100$ in $k = 500$, ter povečati število simulacij. Tega nisem storil, saj je moj računalnik že tako ogromno časa porabil za reševanje 100 iger `Simulatorja(500)`.

Kot sem že prej poudaril, je po uspešnosti `Dynamic simulator` najbližje `Simulatorju(500)`, a je precej hitrejši. `Dynamic simulator` bi lahko bil prilagojen tako, da bi igro reševal bolje, vendar sem se zadovoljil z 92% uspešnostjo in zelo hitrim reševanjem.

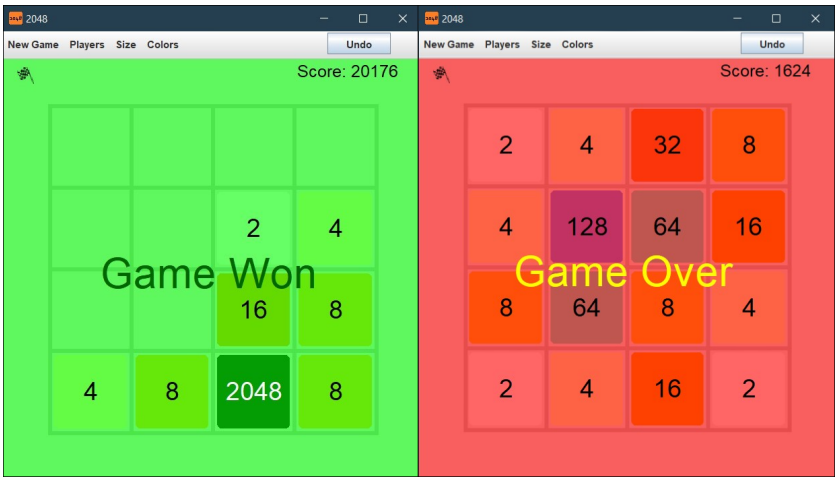
7 Posnetki zaslona



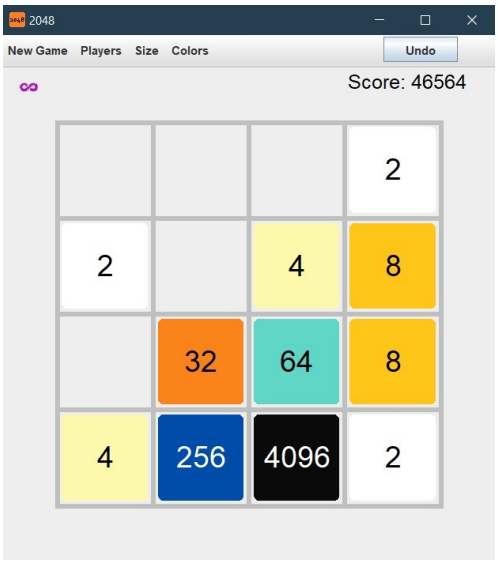
Slika 5: Temna tema in izbiranje računalnikovega algoritma



Slika 6: Igra velikosti 5x5 v *Endless* načinu (ikona zgoraj levo)



Slika 7: Posnetek okna v primeru zmage (levo) in v primeru poraza (desno)



Slika 8: V *Endless* načinu lahko dosežemo tudi večja števila kot 2048

8 Zaključek

Cilj te naloge je bilo ustvariti delujočo aplikacijo, ki ponuja uporabniško izkušnjo, ter nekaj različno dobrih in hitrih računalniških algoritmov. Mislim, da mi je to do neke mere uspelo, algoritem `Dynamic simulator` je dokaj hiter in dokaj uspešen. Če bi želeli še izboljšati uspešnost, bi lahko povečali k v algoritmu `Simulator(k)`. Na primer, vemo, da algoritem vedno premaga igro za $k = 10000$

[3], vendar nihče noče čakati več sekund za posamezno potezo, saj je to zelo dolgočasno. Igra potem traja več kot pol ure. Zato sem se zadovoljil z zelo verjetno zmago in raje hitrejšim časom. Glede izboljšav aplikacije sem lahko najbolj kritičen glede uporabniške izkušnje. Igra bi precej bolje izgledala, če bi vsaka poteza imela animacijo, kot na primer originalna igra Gabrieleja Cirullija, kjer se številke premaknejo iz začetnega mesta pred potezo na končno mesto po potezi. Vseeno sem ocenil, da bi kaj takšnega vzelo preveč čas in sem se zato raje osredotočil na algoritme, analizo in korektno delovanje aplikacije.

Glede algoritmov bi lahko dodal tudi kakšnega, ki uporablja evalvacijo pozicije. Tega nisem storil, saj bi bil algoritem, ki deluje na pozicijski bazi, neuporaben za mrežo različnih velikosti. Zato sem se raje osredotočil na Monte Carlo algoritme, je pa treba poudariti, da s povečanjem mreže algoritmi rabijo precej več časa za izračun poteze. Igro namreč `Random moves` ne izgubi tako hitro in simulacije vzamejo več časa.

Viri in literatura

[1] Wikipedia. 2048 (video game) — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=2048%20(video%20game)&oldid=1083118206`, 2022. [Online; dostopano 25.4.2022].

[2] Ahmed Abdelkader, Aditya Acharya, and Philip Dasler. On the complexity of slide-and-merge games. *CoRR*, abs/1501.03837, 2015.

[3] Ronenz (https://stackoverflow.com/users/632039/ronenz). What is the optimal algorithm for the game 2048? `https://stackoverflow.com/a/23853848/12231075`, 2018. [Online; dostopano 25.4.2022].