Game of Hard Life

Cellular agents, Pattern recognition, Emergence

TFG - Intel.ligencia Artificial

Andreu Bió Llabrés *

Curs 2022/23
Semestre 2

*IDP: 1061410

Llicència

Fitxa del treball final

Títol	*Game of Hard Life*
Paraules clau	*Cellular agents*, *Pattern recognition*, *Emergence*
Autor	Andreu Bió Llabrés
Consultor	Dr. David Isern Alarcón
PRA	Dr. Xavier Baró Solé
Data de lliurament	20 de Juny de 2023
Titulació	Grau en Enginyeria Informàtica
Àrea del Treball Final	05.629 · TFG - Intel.ligencia Artificial
Nombre de crèdits	12
Idioma	Català

Resum del Treball

En aquest TFG es pretén analitzar els sistemes complexes basats en una implementació del reconegut *Game of life*, dissenyat originalment pel matemàtic *John Horton Conway*, per fer-ne una extensió sobre les seves condicions, creant una variant on es puguin considerar situacions més complexes, aplicar-hi algorismes de reconeixement de patrons i un sistema de presa de decisions per tal d'analitzar les conseqüències dels sistemes emergents resultants.

Abstract

In this FDP it is intended analyse complex systems based on a basic implementation of the well-known *Game of life*, originally devised by the mathematician *John Horton Conway*, to expand its conditions and create a variant where more complex situations can be considered, apply pattern recognition algorithms and a decision-making system in order to analyse the consequences of the resulting emergent systems.

Agraïments

A mode personal, m'agradaria que la modesta aportació d'aquest treball pugui servir com a commemoració al reconegut matemàtic britànic *John Horton Conway*, qui a més de dissenyar el plantejament original del *Game of Life*, ha estat un prolífic investigador dins les àrees de teoria de jocs combinatoris, geometria, topologia, teoria de nombres, teoria de grups, àlgebra, anàlisi, algorismes i física teòrica.

John Conway va acabar el seu joc de la vida el 2020 a causa de complicacions per la COVID-19.

Índex

Índex de figures

Índex de taules

1 Resum

Game of Hard Life

Keywords | *Game of life, John Conway, Artificial Intelligence, Automata, Cellular agents, Pattern recognition, Self-organization, Evolutionary game theory, Evolutionary computation, Population dynamics, Decision problem, Machine learning, Complex systems, Emergence.*

L'estudi dels sistemes emergents comporta l'anàlisi de les propietats que es poden observar sobre un sistema donat que no s'hi troben a la naturalesa dels components que ho formen, sinó a conseqüència de les interaccions que es donen entre aquests.

Game of life és un exemple d'entorn en el qual una col · lecció d'entitats relativament simples poden evolucionar en comportaments que sorgeixen a partir de les seves interaccions, i observar com es poden formar grups i components als quals podem reconèixer conceptes com són la translació, creixement, producció o cicles periòdics, que als seus individus no només no se'ls pot aplicar, sinó que no tenen cap disseny associat a aquestes propietats.

Aquest experiment es va dissenyar originalment pel reconegut matemàtic britànic John Horton Conway a l'any 1970. Des de la seva publicació s'han desenvolupat nombroses variants, ja sia modificant el comportament dels autòmats cel · lulars, les condicions de despoblació/sobrepoblació, o les propietats de l'entorn modificant la graella quadriculada bidimensional en una en hexagonal, tridimensional, o projectada sobre una superfície toroidal.

En aquest TFG es pretén partir de la implementació original per aplicar-hi noves variants i aplicar-hi sistemes de reconeixement i decisió per aconseguir que es pugui categoritzar automàticament l'aparició d'aquests patrons i que un agent dedicat a procurar maximitzar la població pugui aplicar noves polítiques de població.

Aquest model de reconeixement podria ser transportable a l'anàlisi de poblacions de formes bàsiques d'organització de sistemes multi-agent, siguin entitats virtuals o representacions de sistemes complexes reals.

El producte final seria la implementació d'aquesta variant i d'un agent amb capacitat de reconeixement de patrons i de decisió. Però posteriorment, a partir d'aquest producte també serà interessant analitzar els nous patrons que puguin aparèixer, o si sorgeixen noves construccions notables.

Com a cloenda, pot ser important investigar si aquest agent de control de polítiques de població podria ser extrapolable a altres camps més enllà d'aquest

experiment, com podria ser dins un sistema robotic o anàlisis de patrons microbiològics.

1.1 Antecedents

La primera presentació pública del *Conway's Game of Life* es mostrava com una primera resposta a un problema presentat durant la dècada de 1940 pel reconegut matemàtic John von Neumann, que pretenia trobar una màquina hipotètica que pogués construir còpies de si mateix.

Des d'aquest punt, la manera en la qual es generen els patrons han despertat molt d'interès des de diverses branques o àrees d'estudi: matemàtics, físics, biòlegs, economistes i analistes computacionals han trobat diferents formes en què emergeixen sistemes complexos a partir d'unes regles tan simples.

Durant els primers anys es van catalogar els primers oscil · ladors, gilders i naus, i als anys '80 John Conway demostra que amb aquests components es pot construir una màquina Turing i també es desenvolupa software específic per fer simulacions.

En les següents decades es van descobrint noves formes més complexes, es desenvolupen programes més complets i es dona d'alta la LifeWiki[1] amb una col · lecció de més de 600 patrons durant l'any 2009.

Dins aquesta darrera dècada, encara hi ha una important activitat d'investigació, i es publica l'any 2022 el llibre: Conway's Game of Life - Mathematics and Construction, de Nathaniel Johnston i Dave Greene

També aquest any es resol el problema que va plantejar John Conway l'any 1972 conegut com el problema del pare únic, obtenint el primer *still life* no sintetitzable que no es pot construir de cap manera des d'altres components.

El projecte ha tingut un interès creixent fins al dia d'avui, encara que sigui com exercici teòric, des de diferents branques del coneixement.

1.2 Mètode

Es pretén implementar una variant del programa *Conway's Game of Life* per aplicar les condicions característiques que es pretenen analitzar en aquest treball.

El programa s'escriurà en codi Python, essent un llenguatge molt dinàmic i amb gran popularitat dins de l'àmbit de la intel · ligencia artificial i la ciència de dades.

Per a dur a terme certes pràctiques a l'hora d'elaborar el codi, s'utilitzarà un projecte Git vinculat a la plataforma de GitLab[2]. A més, en aquesta plataforma es permet fer un panell *Kanban* per fer el seguiment de les tasques i objectius planejats.

[1] LifeWiki [En línia]. LifeWiki (2023). Disponible a: https://conwaylife.com/wiki
[2] GitLab [En línia] (2023). Disponible a: https://gitlab.com

Finalment, analitzar amb el nou programa els possibles comportaments emergents, ja sigui a partir de l'exploració ajustant les condicions inicials, o a través d'un component automatic de generació de casos i simulacions.

1.3 Resultats

Versió inicial del programa *Game of Hard Life*. El codi es troba disponible entre els annexos i també en línia de forma pública al portal de GitLab:

Game of Hard Life
https://gitlab.com/andreu.bio/game-of-hard-life

També s'han analitzat a partir de les fases d'anàlisi i experimentació els primers comportaments emergents dins d'aquesta varietat de la simulació, tractats a la corresponent secció 5.4 Simulacres.

1.4 Aportació i conclusions

S'ha aconseguit elaborar la versió inicial del programa que reprodueix el model de la varietat *Game of Hard Life*, i s'han pogut analitzar alguns dels primers comportaments emergents que han sorgit durant l'experimentació. Aquesta varietat pot passar a formar part del gran ecosistema que ha acumulat el projecte *Conway's Game of Life* dins la seva llarga història.

Addicionalment, aquest treball ha servit per descobrir pràctiques i estratègies d'optimització que han resultat molt interessants i enriquidores, que són extrapolables a pràcticament qualsevol àmbit de la programació.

2 Introducció

2.1 Context i justificació del Treball

Des de la primera publicació del *Game of Life* del matemàtic John Conway al 1970, el que havia començat com un experiment teòric en resposta a una qüestió plantejada per John von Newman sobre màquines auto-replicants, s'han fet diversos avanços tant sobre el model teòric, les eines de software i el descobriment de patrons o figures amb propietats notables.

Tant és així que encara en l'actualitat sorgeixen noves troballes que responen preguntes obertes des dels seus inicis, i no només això, sinó que també han sorgit tota una col · lecció de variants que aporten diferents propietats o amplien la complexitat de l'experiment.

Tot i així, gran part dels descobriments solen tractar-se de trobar certs patrons que desencadenen en certs comportaments o propietats, però no deixen de ser construccions.

En aquest treball es planteja afegir una variant on es té en compta la mortalitat dels automates cel · lulars, i es pretén fer una recerca no tan enfocada en la de trobar figures notables, sinó de comportaments que puguin sorgir de manera *natural* a partir de mostres de població aleatòria.

Així doncs, la varietat *Game of Hard Life* aporta una nova condició que modifica el comportament dels sistemes emergents i es cerca nous patrons o comportaments interessants que puguin sorgir d'aquesta versió.

2.2 Objectius del Treball

En aquest TFG s'utilitza el *Game of Life* per a l'estudi de sistemes emergents sobre aquest entorn i profunditzar en la generació de nous escenaris que puguin sorgir a partir de certes variacions d'aquest simulador de vida virtual.

Per tant, tenim un component analític i d'interpretació dels comportaments d'aquests sistemes evolutius generats per les interrelacions dels automates cel · lulars, i un altre component tècnic de disseny del programari que permeti generar aquest entorn, així com donar la flexibilitat suficient per aplicar-hi les variacions amb les quals es pretén experimentar.

En el *Game of Life* original, a partir d'una població aleatòria, se sol arribar en els primers períodes a una situació majoritàriament fixa, dominada per petits grups que formen components estables fixats o periòdics, deixant unes poques àrees de relativa activitat.

Normalment, s'acaba obtenint un estat de petites agrupacions estables, siguin fixes o de períodes curts. Per donar-li major realisme, una de les varietats que es pretén aplicar és la d'assignar una certa caducitat a les cel · les vives. Per contrarestar la tendència a la mortalitat general, es vol dissenyar un agent

que analitza la situació global, realitzi prediccions i que tingui la capacitat de modificar les condicions de control de població.

Aquest nou agent, a partir de la informació de l'estat del seu univers, haurà de resoldre quina nova política cal aplicar per mantenir la supervivència i tractar d'optimitzar o garantir la vida.

Es divideixen dues fases que en línies generals i que podem resumir en:

Fase 1: Tractament i anàlisi i aplicació de l'entorn *Game of Life* basic.

Fase 2: Desenvolupament de varietats, experimentació i anàlisi aplicant intel · ligència artificial.

En general hi ha els següents tipus d'objectius:

- Documentació
- Anàlisi: obtenció de dades
- Anàlisi: resultats
- Desenvolupament

2.3 Enfocament i mètode seguit

Es vol recrear un simulador de tipus *Conway's Game of Life* on s'aplica una variant que modifica les condicions de vida afegint un component de caducitat als automates cel · lulars.

S'ha optat per produir un programa des de cero per aplicar aquesta varietat, per una banda perquè suposa un canvi important en el plantejament del model típic on els valors binaris d'estat viu/mort passen a tenir valors enters; per altre banda, aquest exercici és en sí una expressió de l'interés humà en desenvolupar estructures complexes a partir de regles simples, i és interessant precisament veure les dificultats que poden sorgir quan es van incorporant nous components i varietats.

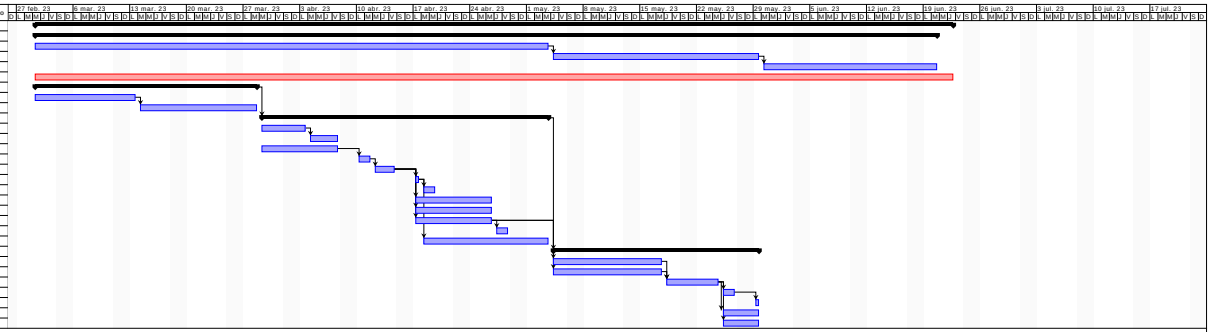
La intenció no és tant la de generar un producte que pugui competir amb les eines ja existents després de 50 anys d'evolució, com la de redescobrir una nova forma d'interpretar el plantejament, solventar possibles complicacions posant en pràctica els coneixements adquirits, i descobrir si aquestes noves condicions poden aportar una revisió de nous patrons de comportament d'aquests automates cel · lulars simples.

2.4 Planificació del Treball

Codi	Objectiu	Requisit	Data límit
I-01	Definició inicial del projecte	∅	13/03
I-02	Pla de treball	∅	28/03
A-01	Recopilació de dades experimentals a partir de l'entorn base *Game of Life*	∅	03/04
A-02	Anàlisi de les dades experimentals A-01	A-01	07/04
G-01	Preparació de l'entorn de desenvolupament	∅	07/04
G-02	Desenvolupament de l'entorn inicial *Game of Life*	G-01	11/04
G-03	Versió inicial de *Game of Hard Life*	G-02	14/04
A-03	Recopilació de dades experimentals a partir de l'entorn *Game of Hard Life - betta*	G-03	17/04
A-04	Anàlisi de les dades experimentals A-03	A-03	19/04
G-04	Varietat: activació d'entorn toroidal	G-03	26/04
G-05	Varietat: Configuració de condicions de vida o mort	G-03	26/04
G-06	Varietat: Configuració de la caducitat de les caselles	G-03	26/04
A-05	Recopilació de dades experimentals a partir de l'entorn *Game of Hard Life - 0.3*	G-06	28/04
A-06	Anàlisi de les dades experimentals A-05	A-05	03/05
D-01	Memòria del TFG (Fase 1)	∅	03/05
G-07	API d'obtenció de metadades	G-06	17/05
G-08	API de control d'entorn	G-06	17/05
G-09	Agent de control de població	G-07/8	24/05
A-07	Recopilació de dades experimentals a partir de l'entorn *Game of Hard Life - 1.0*	G-09	26/05
A-08	Anàlisi de les dades experimentals A-07	A-07	29/05
D-02	Memòria del TFG (Fase 2)	D-01	29/05
G-10	Agent de control amb gestió de recursos limitats	G-09	29/05
G-11	Reconeixement de patrons	G-09	29/05
D-03	Memòria del TFG (**Final)**	D-02	20/06
D-04	Elaboració de la presentació	∅	25/06
	Fase Final		
A-09	Anàlisi de comportaments emergents	A-08	20/06
D-3	Memòria del TFG (**Final)**	D-02	20/06

Taula 1: Pla de Treball

		Nombre	Duracion	Inicio	Terminado	Predecesores	Nombres del Recurso
1		**TFG - Game of Hard Life**	**82 days?**	**1/3/23 8:00**	**22/6/23 17:00**		
2		**Memòria del TFG**	**80 days?**	**1/3/23 8:00**	**20/6/23 17:00**		
3		Memòria del TFG (Fase 1)	46 days?	1/3/23 8:00	3/5/23 17:00		
4		Memòria del TFG (Fase 2)	18 days?	4/5/23 8:00	29/5/23 17:00	3	
5		Memòria del TFG (Final)	16 days?	30/5/23 8:00	20/6/23 17:00	4	
6		Elaboració de la presentació	82 days?	1/3/23 8:00	22/6/23 17:00		
7		**Fase inicial**	**20 days?**	**1/3/23 8:00**	**28/3/23 17:00**		
8		Definició inicial del projecte	9 days?	1/3/23 8:00	13/3/23 17:00		
9		Pla de treball	11 days?	14/3/23 8:00	28/3/23 17:00	8	
10		**Fase 1**	**26 days?**	**29/3/23 8:00**	**3/5/23 17:00**	**7**	
11		Recopilació de dades experime...	4 days?	29/3/23 8:00	3/4/23 17:00		
12		Anàlisi de les dades experimen...	4 days?	4/4/23 8:00	7/4/23 17:00	11	
13		Preparació de l'entorn de dese...	8 days?	29/3/23 8:00	7/4/23 17:00		
14		Desenvolupament de l'entorn i...	2 days?	10/4/23 8:00	11/4/23 17:00	13	
15		Versió inicial de Game of Hard L...	3 days?	12/4/23 8:00	14/4/23 17:00	14	
16		Recopilació de dades experime...	1 day?	17/4/23 8:00	17/4/23 17:00	15	
17		Anàlisi de les dades experimen...	2 days?	18/4/23 8:00	19/4/23 17:00	16	
18		Varietat: activació d'entorn tor...	8 days?	17/4/23 8:00	26/4/23 17:00	15	
19		Varietat: Configuració de condi...	8 days?	17/4/23 8:00	26/4/23 17:00	15	
20		Varietat: Configuració de la cad...	8 days?	17/4/23 8:00	26/4/23 17:00	15	
21		Recopilació de dades experime...	2 days?	27/4/23 8:00	28/4/23 17:00	20	
22		Anàlisi de les dades experimen...	12 days?	18/4/23 8:00	3/5/23 17:00	16	
23		**Fase 2**	**18 days?**	**4/5/23 8:00**	**29/5/23 17:00**	**10**	
24		API d'obtenció de metadades	10 days?	4/5/23 8:00	17/5/23 17:00	20	
25		API de control d'entorn	10 days?	4/5/23 8:00	17/5/23 17:00	20	
26		Agent de control de població	5 days?	18/5/23 8:00	24/5/23 17:00	24;25	
27		Recopilació de dades experime...	2 days?	25/5/23 8:00	26/5/23 17:00	26	
28		Anàlisi de les dades experimen...	1 day?	29/5/23 8:00	29/5/23 17:00	27	
29		Agent de control amb gestió d...	3 days?	24/5/23 17:00	29/5/23 17:00	26	
30		Reconeixement de patrons	3 days?	25/5/23 8:00	29/5/23 17:00	26	



2.5 Breu sumari de contribucions i productes obtinguts

Components principals entregats:

- Game of Hard Life - v1

  Contingut del projecte git adjunt a l'annexe `game-of-hard-life.zip`, i que representa el producte principal d'aquest treball.

  També es troba disponible en línia de forma pública al portal de GitLab: Game of Hard Life (https://gitlab.com/andreu.bio/game-of-hard-life)

- Anàlisi de comportaments emergents

  Anàlisi dels primers patrons reconeguts de comportament global, descrits a la secció 5.4 Simulacres, en es presenten a més de la descripció d'algunes configuracions d'interès, la classificació dels patrons globals de tipus *polse* i *invasiu*.

- Dades experimentals

  Col · lecció de mostres a partir de la recopilació de dades experimentals durant les diferents fases.

- Document: Seguiment del treball

  Seguiment complet de l'evolució del projecte amb la descripció de tasques planificades i la descripció d'activitats no previstes o mitigació de les seves desviacions en la temporalització.

2.6 Breu descripció dels altres capítols de la memòria

Els següents capítols detallen els diferents apartats i contextos de la memòria:

3 Estat de l'art: Resum de les principals eines, programes o recursos que existeixen actualment sobre el *Conway's Game of Life*

4 Metodologia: S'indica la classificació dels tipus generals de tasques i objectius, i l'organització del *workflow* del projecte

5 Resultats: Es mostren els resultats més rellevants de cada categoria de tasques, descrivint els punts importants dels diferents apartats de manera analítica i exposant part de les mostres o del producte que s'està tractant.

6 Discussió: Es dona una visió global dels resultats finals dins el context del projecte, i es valora si s'han aconseguit els objectius proposats.

7 Conclusions: Descripció de les conclusions i aprenentatges al llarg del treball i revisió de l'assoliment d'objectius o presentació de línies a futur del projecte.

3 Estat de l'art

Com s'ha explicat al punt 1.1 Antecedents, *Conway's Game of Life* acumula ja 50 anys de desenvolupament i descobriment de figures, patrons i propietats que es poden trobar a conseqüència dels sistemes emergents a partir de les simples regles que ho componen.

En aquesta secció, es recullen alguns dels resultats o exemples més notables que hi ha treballats fins a l'actualitat:

3.1 Simuladors a la web

Life Viewer[3]

Aquest és el simulador més competent i robust que s'ha pogut trobar disponible en línia a la Web. A més d'opcions de visualització, presentació, control de velocitat de generacions, zoom i ajust de pantalla a la figura, disposa també de multitud d'eines d'intervenció o generació de patrons, així com d'importació/exportació d'un cert estat seguint les nomenclatures estandaritzades de *Rulestring*.

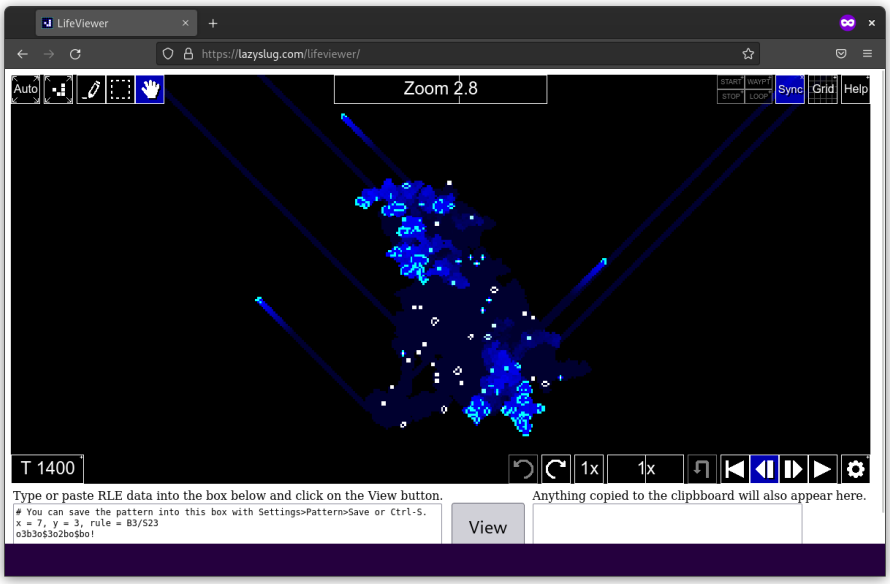


Figura 1: LazySlug LifeViewer - Simulació inicial t:1400

Hi ha molts altres exemples disponibles a la web, encara que són models més simples i bàsics, pero poden aportar alguna novetat com el model toroidal, com

[3]Life Viewer [En línia] (2023). Disponible a: https://lazyslug.com/lifeviewer

per exemple el que es troba al portal de beltoforion.de[4]

3.2 Software

Golly[5]

Aquest és possiblement el software més complet tant per l'extens catàleg de figures d'interès com per les possibilitats de configuració i experimentació, seguint els estàndards de definició.

Un gran avantatge d'aquest programa és que aplica algoritmes molt optimitzats com el *HashLife* que eviten que el pas de generacions suposi una complexitat algorísmica de $O(n \times m)$ segons les dimensions m, n de la graella.

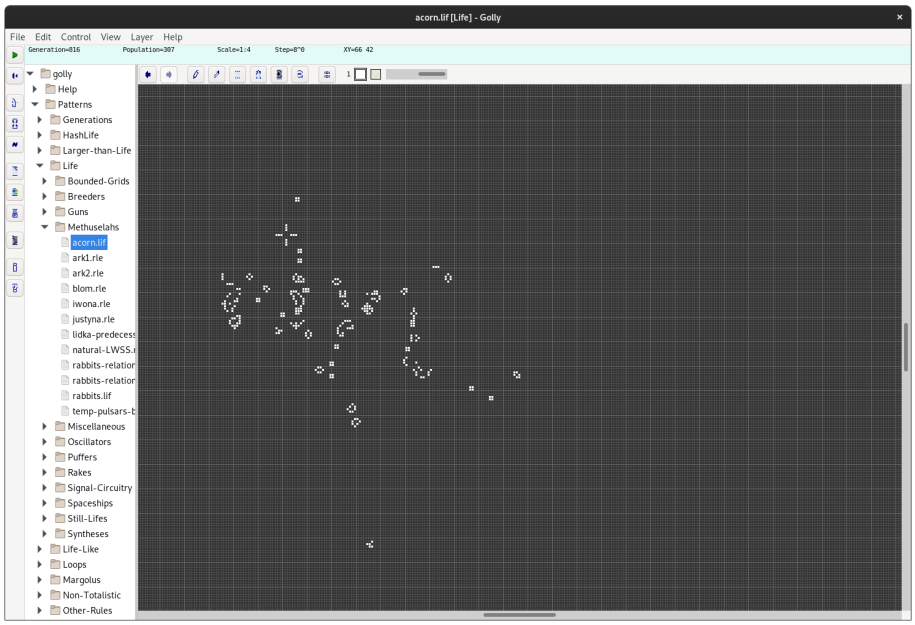


Figura 2: Golly - Simulacre d'execució del *acorn*, gen:816

3.3 Catàleg

Possiblement la documentació més detallada i completa en línia es pot trobar a partir de la LifeWiki[6]. En aquesta s'hi pot trobar tant informació de consulta com recursos externs. Del mateix equip que manté el portal, s'enllaça un extens

[4] John Conway's Game of Life - An Introduction to celluar Automata [En línia] (2023). Disponible a: https://beltoforion.de/en/game_of_life

[5] Golly Game of Life Home Page [En línia] (2023). Disponible a: https://golly.sourceforge.net

[6] LifeWiki [En línia]. LifeWiki (2023). Disponible a: https://conwaylife.com/wiki

catàleg a través del portal hatsya.com[7] iniciat per Adam P. Goucher i operatiu des de 2015, obtingut a partir de l'anàlisi i a partir de processos de cerca automàtica de diferents *soups*, on hi participen centenars de contribuïdors.

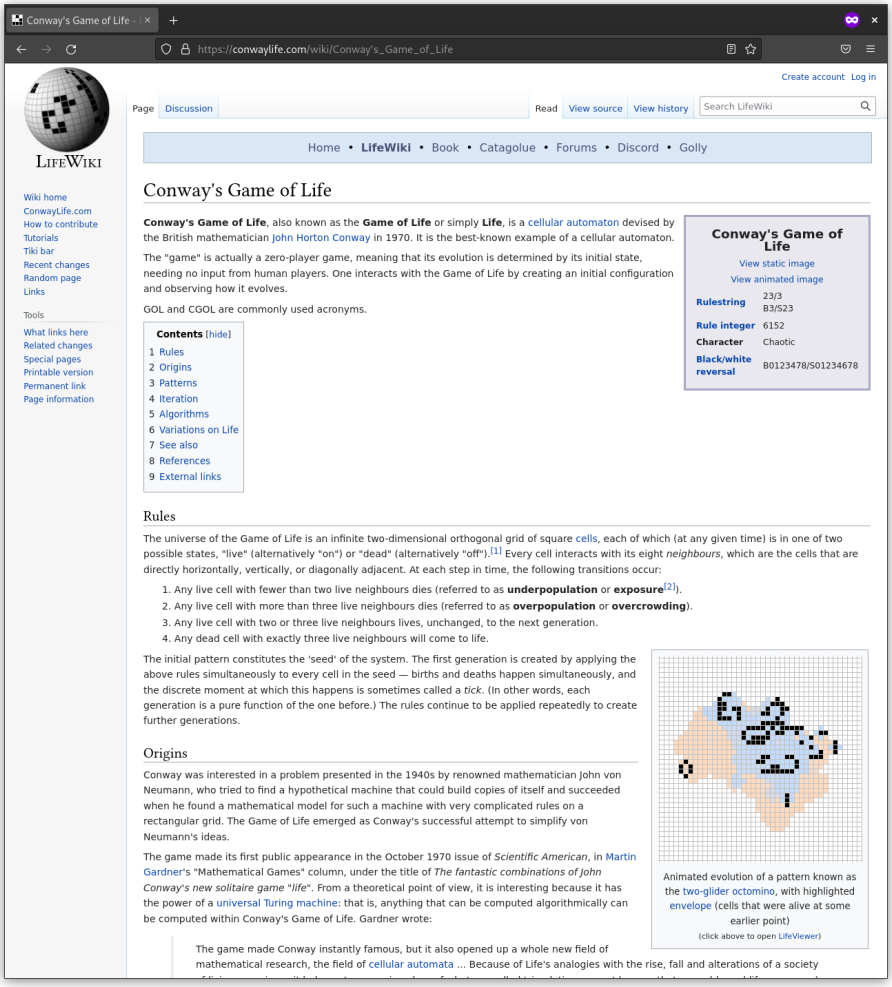


Figura 3: LifeWiki - Pàgina principal

Un exemple és el Census[8] de la regla bàsica (**B3/S23-C1**)on es poden trobar més de 4.7 mil bilions d'objectes catalogats.

[7] Catalogue [En línia]. Adam P. Goucher (2023). Disponible a: https://catagolue.hatsya.com/home

[8] Census - Catalogue [En línia]. Adam P. Goucher (2023). Disponible a: https://catagolue.hatsya.com/census/b3s23/C1

3.4 Variants

Al llarg dels 50 anys d'història han anat sorgint diverses variants i reinterpretacions de l'experiment original. Entre les variants més interessants podem trobar:

3.4.1 Graella hexagonal

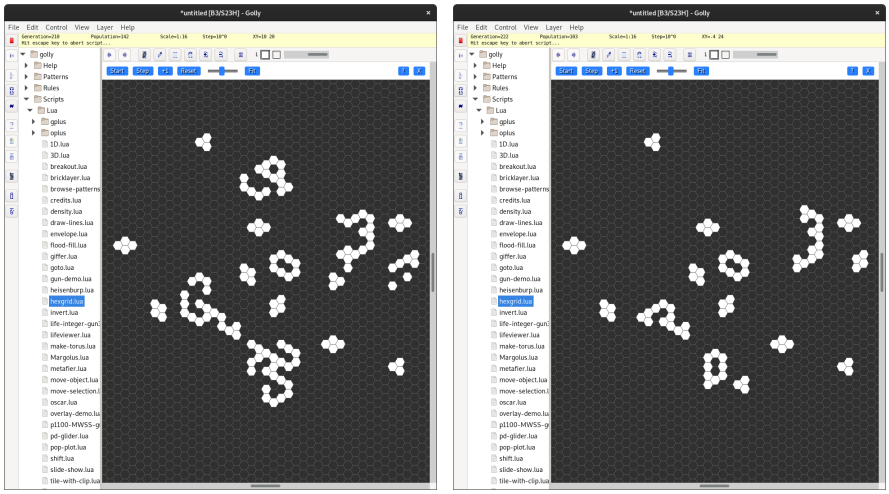


Figura 4: Golly - Variació hexagonal, gen:210/222

Com aquesta, també hi ha casos d'implementació no quadriculades sobre graelles triangulars o pentagonals.

3.4.2 diferents dimensions

Es poden trobar exemples d'aplicació del *Game of Life* en 3D, o també una versió reduïda al cas unidimensional:

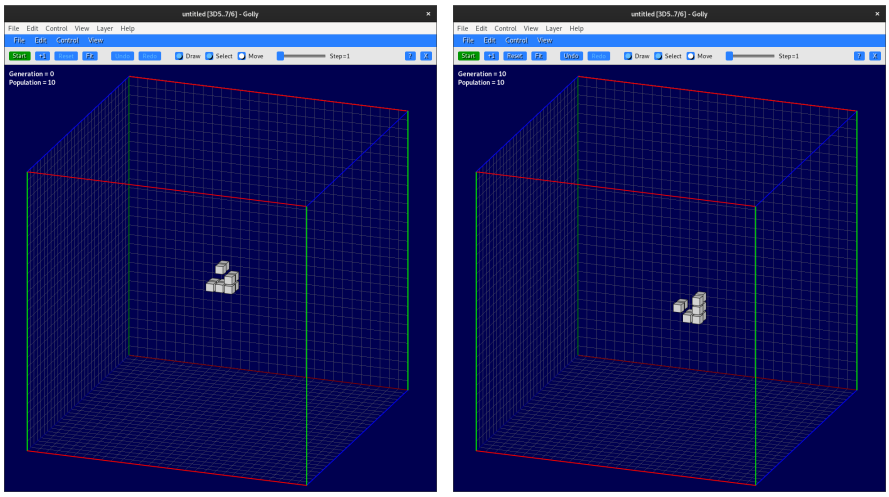


Figura 5: Golly - Variació tridimensional, gen:0/10

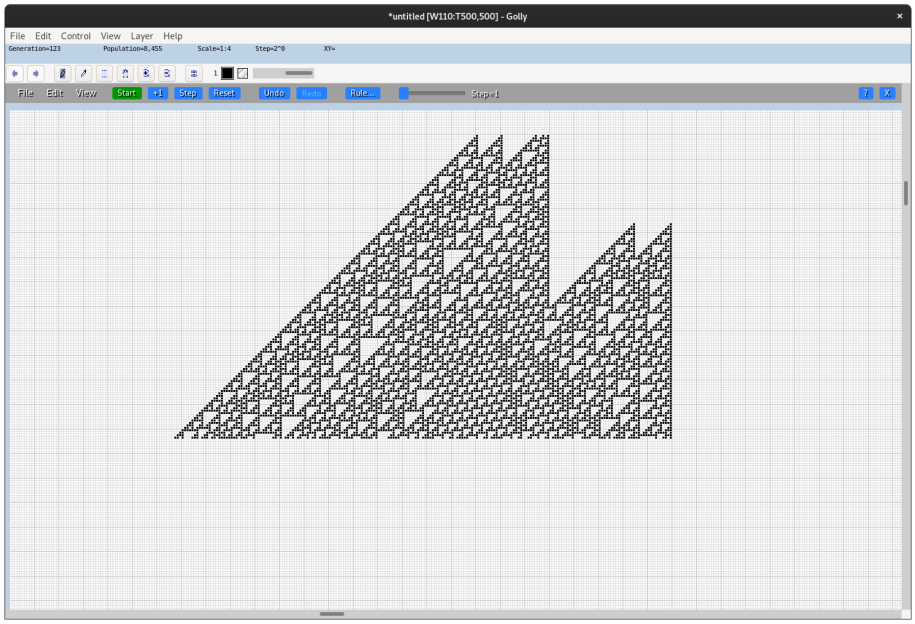


Figura 6: Golly - Variació unidimensional, gen:123

3.4.3 Canvis de regles

Un cop es canvien les regles bàsiques, es poden construir tota classe de figures amb comportaments elaborats i deterministics o bé recreacions caòtiques de tota mena.

Per exemple, per il · lustrar casos constructius es mostren regles on es representa per colors la simulació d'un circuit elèctric, o una calculadora de nombres primers:

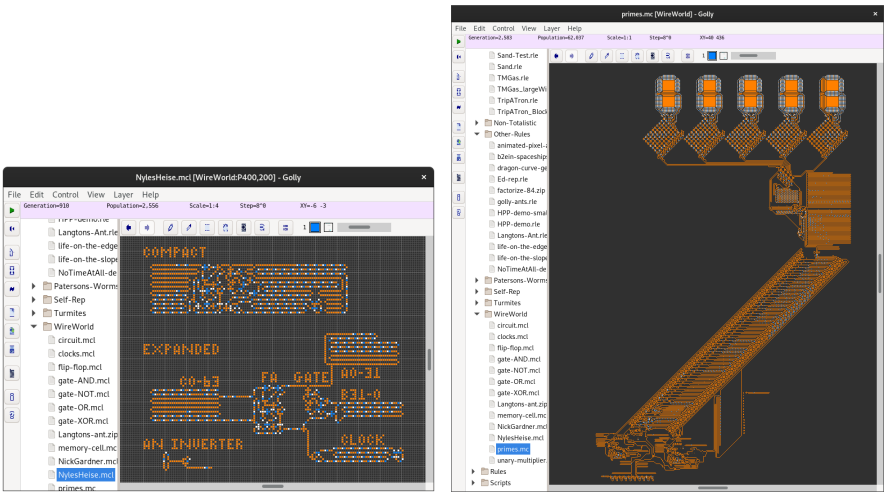


Figura 7: Golly - Variació de regles, circuit/primers

La creativitat de les contribucions amb llibertat de regles arriba a ser extraordinària, i encara que pugui considerar-se una simple forma d'expressió de creativitat i enginy, el descobriment de nous patrons és en si mateix un exercici d'interès intel · lectual.

4 Metodologia

Entre les diferents fases generals del projecte, s'han organitzat les tasques i objectius entre aquestes categories:

Analítica: En aquestes tasques s'ha plantejat en diferents entorns, essent aquests externs o bé amb el propi producte *Game of Hard Life* en les seves diferents etapes d'evolució, realitzar l'experimentació d'executar cada simulacre a partir d'un estat homogeni.

La norma és obtenir una graella quadrada de cel · les aleatòries i executar les generacions sense intervenció, per tal d'arribar o bé a alguna situació cíclica, o bé reconèixer una situació que s'estén infinitament sense aportar més interaccions.

En aquestes anàlisis es comprova principalment quines figures es reconeixen del resultat donat de manera natural, i quina és la proporció de població viva respecte al corresponent estat inicial.

Desenvolupament: Aquestes tasques són on es defineixen els objectius que s'espera aconseguir del producte *Game of Hard Life*.

El programa va evolucionant a mesura que s'aplica la implementació de noves funcionalitats o comportaments esperats.

Refinament: Aquestes són tasques que han anat sorgint durant el desenvolupament de l'aplicació, però no han estat motivades per obtenir certa finalitat, sinó a causa d'alguna necessitat que ha sorgit addicionalment durant la implementació d'altres funcions.

L'aplicació d'estratègies *branchless*, l'ús de funcions precompilades o la modificació de la llibreria GUI són els exemples més notables i que s'expliquen més endavant.

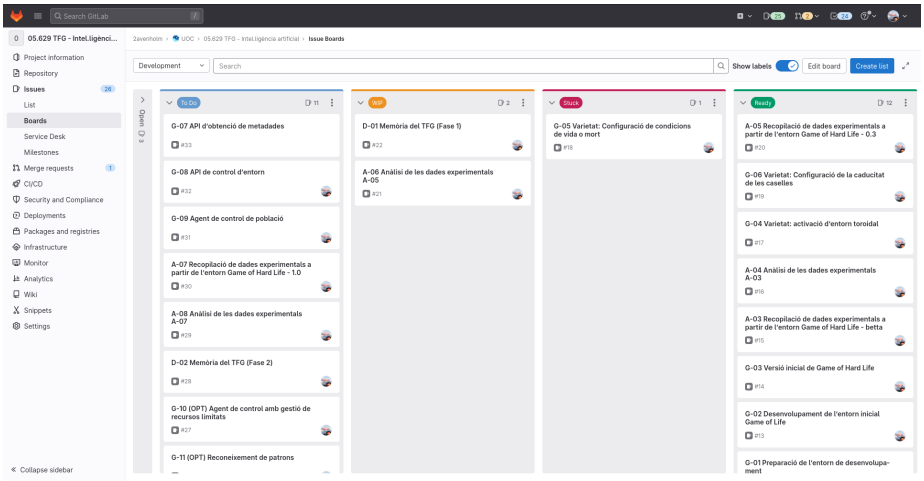
Simulacres: L'execució de simulacres no només comporta un component de la fase analítica, sinó que també són la manera de reconèixer que l'aplicació està funcionant de la manera esperada.

També es mostren els resultats d'execucions de simulacres amb condicions i resultats notables que no parteixen de la situació normalitzada d'iniciar a partir d'una sopa de cel · les aleatòries, que ajuden a entendre millor les repercussions de la variació sobre la qual es vol treballar.

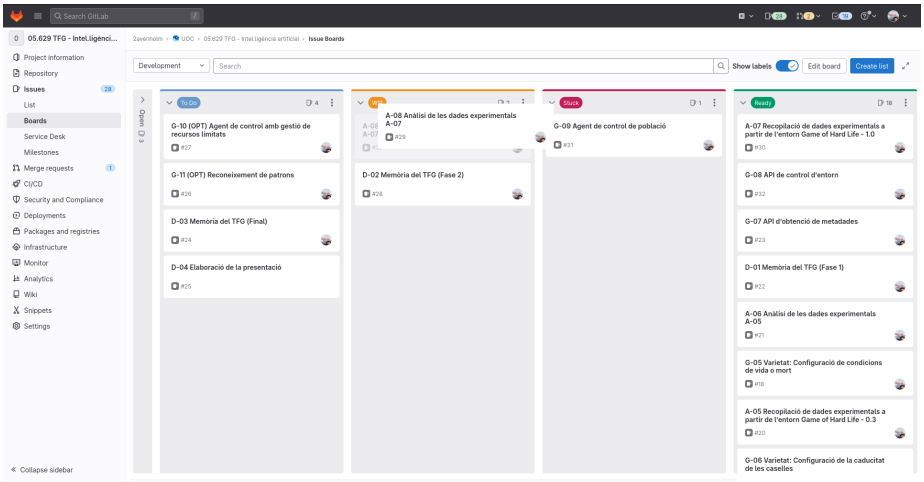
L'organització de les diferents tasques definides al 2.4 Planificació del Treball s'ha manejat a través d'un panell Kanban.

El panell Kanban forma part d'una de les eines disponibles entre els elements organitzatius d'un projecte a GitLab[9]. Per això, aprofitant l'ús del repositori git, s'ha combinat l'organització de les corresponents *Issues* en aquest panell, perseguint així una dinàmica i flux de treball organitzada i de manera visual.

[9] GitLab [En línia] (2023). Disponible a: https://gitlab.com



(a) Kanban - Fase I



(b) Kanban - Fase II

Figura 8: Kanban Board - GitLab

5 Resultats

5.1 Analítica

5.1.1 A-2, Anàlisi sobre l'entorn base *Game of Life*

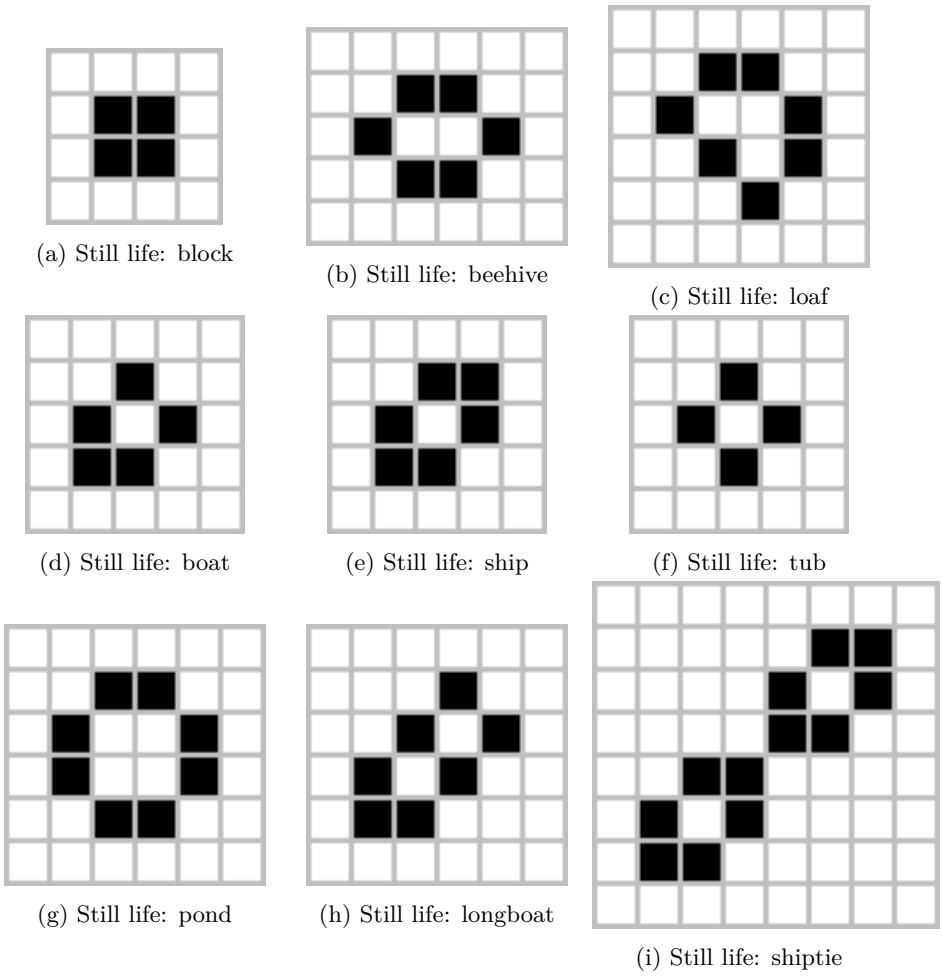
En aquesta secció es presenten els resultats que es poden obtenir a partir de l'entorn base *Game of Life*, iniciant des d'una *soup* generada de manera homogènia a partir d'una regió de cel · les aleatòries.

Amb aquest experiment es tracta de reconèixer i d'identificar els patrons més comuns que apareixen de forma natural.

Hi ha dos grups de captures de les proves experimentals, unes utilitzant l'aplicació web LifeView i les altres a partir de l'aplicació Golly. Es mostren algunes de les captures en cada cas, però hi ha més mostres al fitxer adjunt on es recullen les mostres.

En general, les observacions confirmen l'aparició dels patrons reconeguts a la llista de *still life*[10] més comuns:

[10] List of common still lifes [En línia]. LifeWiki (2023). Disponible a: https://conwaylife.com/wiki/List_of_common_still_lifes



(a) Still life: block

(b) Still life: beehive

(c) Still life: loaf

(d) Still life: boat

(e) Still life: ship

(f) Still life: tub

(g) Still life: pond

(h) Still life: longboat

(i) Still life: shiptie

Figura 9: Patrons *Still life* més comuns

LifeView

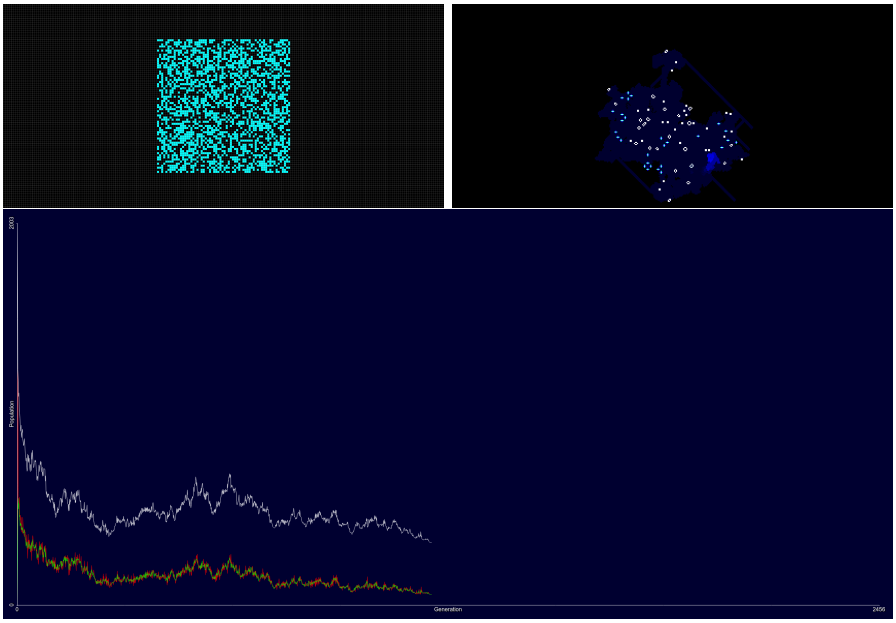


Figura 10: Experiment 0 - LifeView

Sobre aquest entorn s'aplica un estat inicial que forma una graella quadrada de cel · les aleatòries. S'activa una rutina de reconeixement de *gliders* que escapen del mapa a l'infinit i no aporten més interacció, i es deixa reproduir el simulacre fins que es detecta un estat cíclic.

A les captures es mostra doncs, l'estat inicial, l'estat cíclic final, i la gràfica on es mostra l'evolució de la població.

Es pot observar que el resultat final es composa de figures ben reconegudes i catalogades entre les formes de tipus *still life*, *oscilator*, *glider* o similars.

Altres exemples mostren que la tendència a acabar en aquests comportaments és en general el que podem esperar.

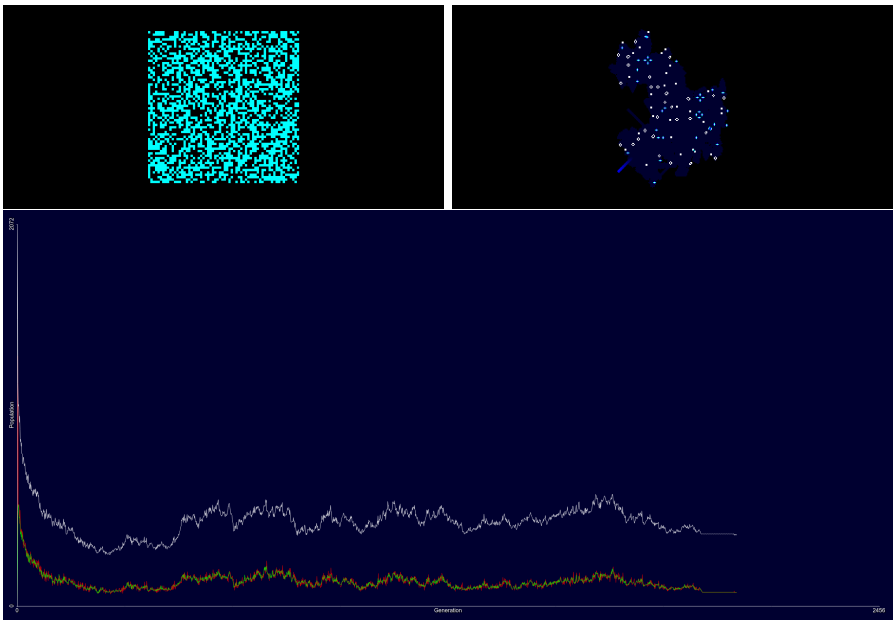
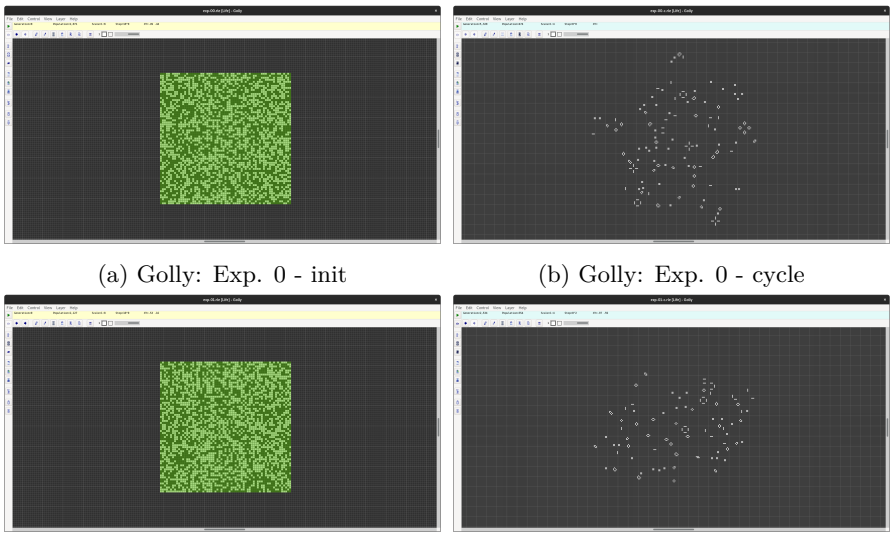


Figura 11: Experiment 1 - LifeView

Golly

S'han realitzat proves similars amb l'aplicació Golly preperant un estat inicial de 32×32 caselles aleatòries i deixant córrer la simulació fins que s'observa un estat cíclic, de nou retirant els *gliders* que estan destinats a no poder interactuar de cap manera possible.



(a) Golly: Exp. 0 - init

(b) Golly: Exp. 0 - cycle

(c) Golly: Exp. 1 - init

(d) Golly: Exp. 1 - cycle

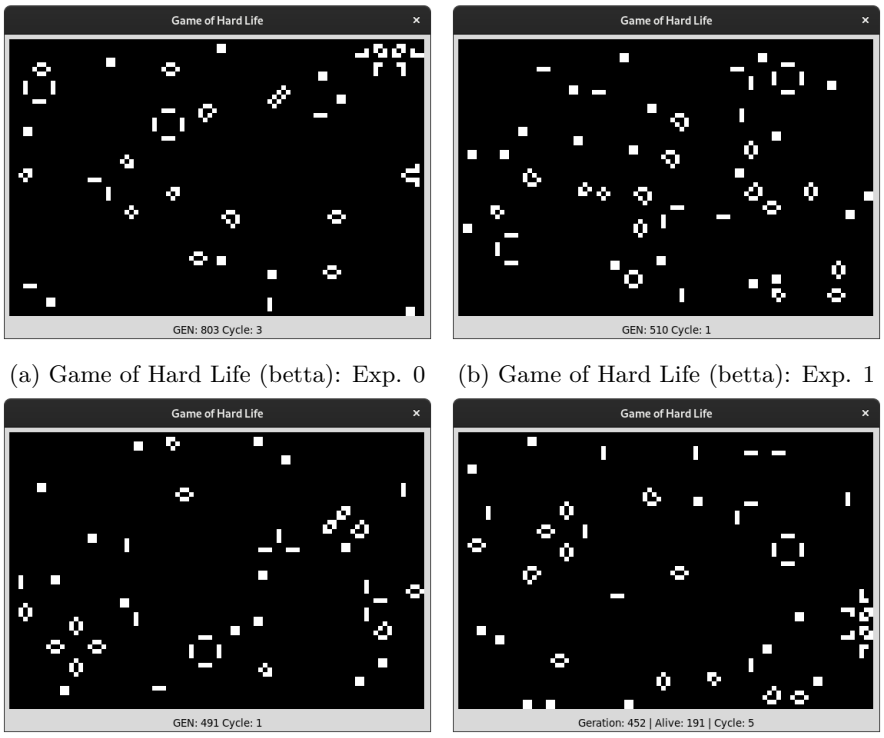
Figura 12: Experiments - Golly

Les observacions experimentals mostren les mateixes tendències on es poden comprobar l'aparició de patrons *still life*, *oscilators* o, tot i que no aparèixer degut a la purga per tal de trobar estats cíclics, de figures de tipus *glider*.

5.1.2 A-4, Anàlisi sobre l'entorn producte: *Game of Hard Life - betta*

Aquesta secció parteix de la primera fase de desenvolupament en la qual s'ha creat la versió primària del simulador que reprodueix les mateixes condicions del *Conway's Game of Life*.

La intenció aquí és, com es pot apreciar, corroborar que partint d'una graella aleatòria, els estats finals acaben presentant les mateixes tendències:



(a) Game of Hard Life (betta): Exp. 0 (b) Game of Hard Life (betta): Exp. 1

(c) Game of Hard Life (betta): Exp. 2 (d) Game of Hard Life (betta): Exp. 3

Figura 13: Experiments - Game of Hard Life (betta)

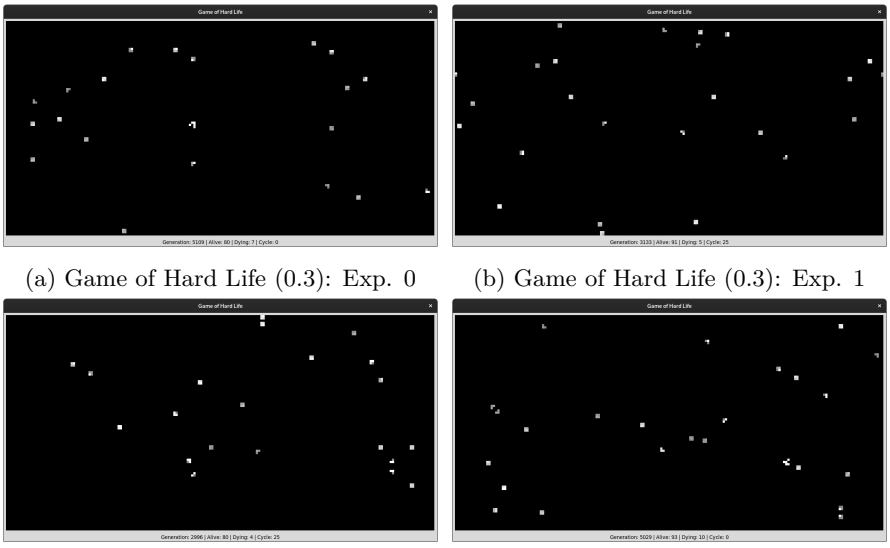
Addicionalment, encara que no es pot apreciar per les captures, l'experiment serveix per validar el reconeixement d'estats cíclics, ja que l'aplicació s'atura automàticament quan es troba en aquest cas.

5.1.3 A-6, Anàlisi sobre l'entorn producte: *Game of Hard Life - 0.3*

En aquesta fase, ja estem utilitzant la varietat pròpia de *Game of Hard Life* on les cel · les tenen una certa caducitat.

Cal afegir que en aquesta versió ja s'aplica el comportament propi de la projecció sobre superfície toroidal.

Com es pot observar, la major part de figures de tipus *still life* ja no es poden mantenir. De fet, l'única que es manté, és el bloc, que ha passat a ser un oscil · lador amb període equivalent al temps de vida màxim de les cel · les.



(a) Game of Hard Life (0.3): Exp. 0 (b) Game of Hard Life (0.3): Exp. 1

(c) Game of Hard Life (0.3): Exp. 2 (d) Game of Hard Life (0.3): Exp. 3

Figura 14: Experiments - Game of Hard Life (0.3)

Les figures de tipus *still life*, en general requereixen en tot moment que cada una de les cel · les es mantingui viva per donar suport a tota l'estructura. El que ocorre en aquesta variant, és que en algún moment una d'aquestes caselles mor, i les demés acaben col · lapsant en un efecte en cadena.

La particularitat del bloc és que de les quatre cel · les que el formen, cada cop que mor una d'elles, les altres tres poden mantenir-se una generació més, i és suficient per a que a la següent es produeixi un nou naixement a la posició de la cel · la que havia passat a estat mort. És per això que el bloc passa de ser un component *still life* estàtic a ser un oscil · lador que es va retroalimentant de forma periòdica.

La figura comú que no presenta canvis és el *glider*, que essent una figura dinàmica, no es veu afectada per la caducitat dels automates.

5.1.4 A-8, Anàlisi sobre l'entorn producte: *Game of Hard Life - 1.0*

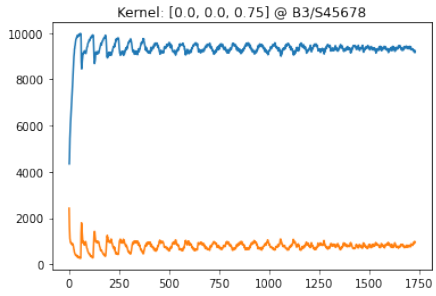
Aquest apartat agafa un àmbit diferent, ja que aquest és la anàlisi dels resultats obtinguts de l'agent, capturant les dades a partir dels YAML que s'han produït llançant les múltiples proves per diferents valors del *Kernel* amb certa *Rulestring*.

La intenció és analitzar directament quines configuracions mostren proporcions de vius i morts que semblin estabilitzar-se, i analitzar després si trobem alguns patrons interessants fent els corresponents simulacres afegint els valors al `settings.py`.

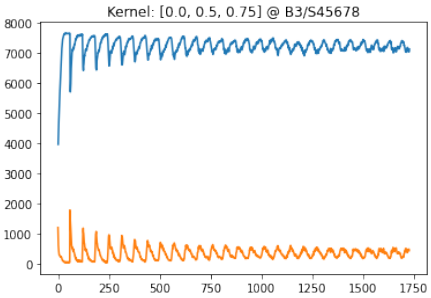
Per realitzar aquest anàlisi, s'ha afegit la llibreta Jupyter `analyze.ipynb`, on s'ha preparat un script simple que recerca a la ruta dels tests els fitxers YAML, importa del dades per cada generació i en mostra les gràfiques evolutives.

La intenció és veure quins valors del *Kernel* mostren una població estable per fer-ne després la simulació i plantejar si es mostren més patrons d'interés.

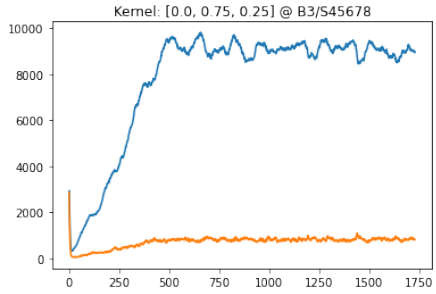
A la següent figura es mostren sis de les gràfiques que poden ser d'interés d'entre les 126 que sorgeixen de la bateria de tests generats per l'agent configurat a realitzar particions de 5 sobre cada component de distancia de Moore del vector.



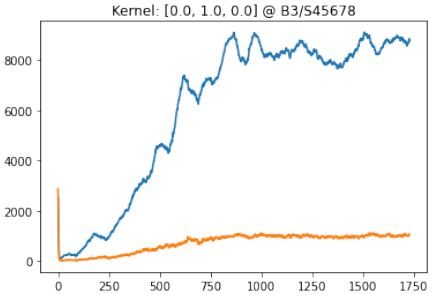
(a) Game Of Hard Life (1.0-Agent): $Kernel = (0, 0, 0.75)$



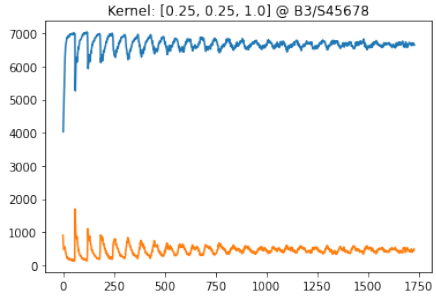
(b) Game Of Hard Life (1.0-Agent): $Kernel = (0, 0.5, 0.75)$



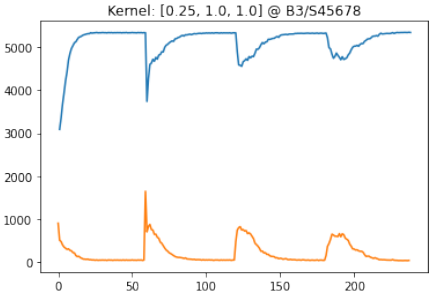
(c) Game Of Hard Life (1.0-Agent): $Kernel = (0, 0.75, 0.25)$



(d) Game Of Hard Life (1.0-Agent): $Kernel = (0, 1, 0)$



(e) Game Of Hard Life (1.0-Agent): $Kernel = (0.25, 0.25, 1)$



(f) Game Of Hard Life (1.0-Agent): $Kernel = (0.25, 1, 0)$

Figura 15: Experiments - Game of Hard Life (Agent)

5.2 Desenvolupament

5.2.1 Estructura del programa

El programa s'ha plantejat en diferents arxius i se separen els components principals per aïllar certes funcionalitats segons el cas d'execució. Es detallen els seus components amb la seva utilitat:

`main.py` És on es defineixen els components principals i on s'hi poden trobar les classes `GameOfHardLife`, `World`, `Registry` i les funcions auxiliars que es troben a l'àmbit global per poder ser precompilades.

`GameOfHardLife`: En aquesta classe es defineix el flux principal del programa per executar-ho amb la interfície gràfica d'usuari (GUI).

Aquesta inicialitza un nou entorn (`world`), defineix la GUI i maneja el control d'execució durant el simulacre que es pot manipular de manera simple amb les tecles d'espai o retorn.

No s'han implementat elements d'interacció, ja que l'objectiu d'aquest treball s'ha centrat en l'anàlisi dels comportaments dels autòmates cel · lulars; però pot ser un dels primers aspectes a millorar entre les línies a futur del projecte.

`World`: Aquesta és la classe principal on hi ha tots els components del simulacre.

Es defineix la topología de la graella, amb les seves dimensions i la definició segons si s'entén sobre un pla o si és la projecció d'una superfície toroidal.

La projecció toroidal planteja que les condicions contemplin que el que es mostren com a límits de la graella rectangular, en realitat formen part d'un cicle complet. Això es tradueix en que el comportament dels automates de les posicions extremes de la graella bidimensional, tinguin en compte com a veïnes aquelles posicions que estan a l'altre costat del mapa.

Si ho representem en tres dimensions obtenim el resultat de la següent figura: *Conway's Game of Life on a Torus*[11]

Dins aquesta classe es controlen els estats de cada generació, l'estat present de la graella de cel · les, el cicle de vida, i alguns paràmetres de control per analitzar els temps de processament de cada generació així com les condicions del comportament depenent del *Kernel* o la *Rulestring*.

També es genera un registre històric per conèixer l'estat en que es trobava en les darreres generacions. Això s'utilitza per a fer el càlcul

[11] Conway's Game of Life on a Torus - *Tim Hutton* [En línia]. Youtube.com (2023). Disponible a: https://www.youtube.com/watch?v=lxIeaotWIks

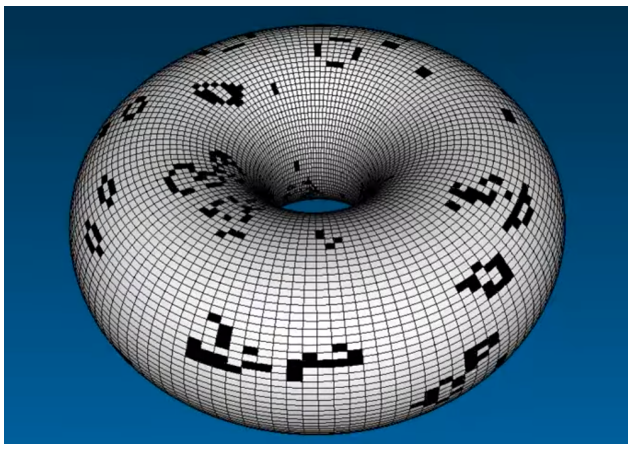


Figura 16: Conway's Game of Life on a Torus - *Tim Hutton*

de l'estat cíclic i determinar si la situació actual és idèntica a una situació anterior, i si podem deduir que el comportament en generacions futures està marcat a ser reiteratiu.

Disposa de tota una col · lecció de funcions per controlar l'estat d'aquest simulacre, exceptuant les dues funcions definides en l'entorn global.

`Registry`: Aquesta classe manté un registre històric de l'estat de la graella fins a un cert límit controlat per variable d'entorn, així com informació sobre les condicions de la simulació i els estats on es guarda la informació de totes les generacions, on s'hi inclou: generació, configuració del *Kernel* i *Rulestring*, nombre de cel · les vives i nombre de les que es preveu que han de morir a la pròxima generació, estat cíclic i el registre dels temps de computació usats.

Aquesta informació s'acaba guardant en format YAML a un fitxer identificatiu del simulacre dins del directori `tests`, per poder utilitzar-ho posteriorment per fer-ne anàlisi.

`ker(), dying_cells()`: Aquestes són funcions que s'han tingut que definir en l'entorn global per a poder ser precompilades a partir de la llibrería `numba`. Es tracta de funcions que s'han d'executar per a cada cel · la, i ha estat crucial fer una crida precompilada per a fer viable l'ús del programa.

`ker()` És la funció destinada a calcular el nombre de veïns a considerar per a una cel · la donada. Normalment, bastaria contemplar les cel · les del voltant i comprovar el seu estat viu/mort, però en aquesta implementació cal considerar l'estat del *Kernel* per veure fins a quina distància es consideren els veïns i quin és el seu pes a l'hora de contabilitzar-los.

`dying_cells()` És una funció molt simple destinada a calcular el nombre de veïns que moren a la pròxima generació, però com s'ha d'executar a cada una d'aquestes generacions també s'hi ha aplicat el mètode de precompilació per accelerar els temps de computació.

`agent.py` En aquest fitxer s'hi importa la classe `World`, pel que es pot realitzar igualment simulacres com ocorre amb el fitxer `main.py`, però s'hi defineixen les funcions necessàries pensades per executar en paral · lel els simulacres experimentals sense presentar cap entorn GUI.

L'agent està pensat per fer diferents particions entre els valors del *Kernel* i executar de forma separada els simulacres a partir de cada una d'aquestes variacions.

Aquest agent està destinat a presentar informació de l'estat de les execucions de cada simulacre a través del propi *stdout* del terminal, però com també obté les funcionalitats del registre, es generen els fitxers YAML amb la informació de cada simulació.

`settings.py` Agrupa les variables globals que configuren l'entorn d'execució del simulacre, essent una forma simple i estructurada de modificar les condicions en les quals es vol experimentar.

`rulestrings.py`,`forms.py` En aquests fitxers es col · leccionen respectivament algunes de les regles o estats inicials més notables, per tal de poder comparar de forma ràpida els comportaments entre aquests casos d'interès.

5.2.2 Descripció del *Kernel* i *Rulestring*

Kernel

El *Kernel* és un vector que representa la consideració que ha de tenir cada cel · la pel que fa a la comptabilització dels veïns del seu entorn a l'hora d'analitzar el seu proper estat segons les condicions de despoblació o sobrepoblació.

Aquest és un vector on s'aplica un coeficient de pes amb relació a la distància de Moore a la que es troben les següents cel · les, essent la consideració de Moore aquelles que es troben a una posició combinant els eixos vertical i horitzontal de la superfície.

Per exemple, les següents figures que representen els rangs 1 i 2 de veïns segons la regla de Moore[12], es corresponen respectivament als vectors *Kernel* $(0, 1)$ i $(0, 1, 1)$:

Aplicar diferents pesos a aquestes distàncies permet alterar les reaccions sobre l'entorn de forma molt diversa, i es pot plantejar acabar obtenint una xarxa neural on es configuren aquests pesos de manera òptima segons la situació donada a una generació.

[12] Moore neighbourhood [En línia]. LifeWiki (2023). Disponible a: https://conwaylife.com/wiki/Moore_neighbourhood

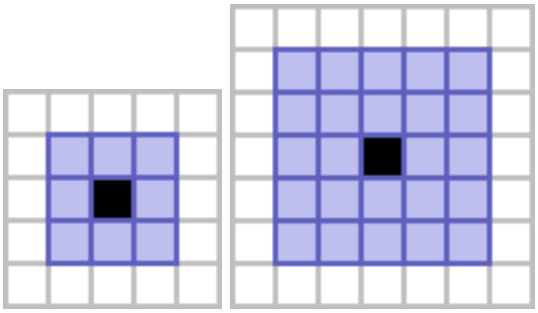


Figura 17: Veïns Moore de rangs 1 i 2

Fins al moment, s'ha analitzat els comportaments que sorgeixen amb diferents configuracions, com per exemple la composició Gaussiana $(0, 0.136, 0.6828, 0.136)$, i s'han aconseguit algunes combinacions de pesos d'interès que es mostren a la secció 5.4 Simulacres

Rulestring

La *Rulestring*[13] és un standard de notació en format *string* amb la qual es defineix el comportament dels automates cel · lulars segons la informació obtinguda del seu entorn.

La notació més comuna és la B/S (*Birth/Survival*), on es concatenen a la primera secció el nombre de veïns vius que compten per considerar un naixement, seguit de la secció on es concatenen els sobres de veïns necessaris per a sobreviure.

Per exemple, la regla clàssica del *Conway's Game of Life* és **B3/S23**, ja que es necessiten tres veïns per considerar un naixement, i ha d'haver-hi dos o tres veïns per tal que una cel · la viva sobrevisqui a la següent generació.

5.2.3 Interfície gràfica d'usuari

La interfície gràfica d'usuari es construeix a partir de la llibrería Tkinter[14] , que és la llibrería per defecte de *Python* per les eines gràfiques Tk.

Per a representar la graella de cel · les de la classe *World* corresponent al simulacre en una imatge, s'utilitza la llibreria PIL (*Python Imaging Library*). Usant les definicions de `Image` i `ImageTk`, es transforma les dades de la matriu on es determinen els estats de les cel · les a valors dels píxels dins un gradient a l'escala de grisos.

Al final, la pantalla que es mostra és aquesta representació gràfica de la matriu de la generació actual, i una barra addicional informativa amb dades de l'estat de la generació.

[13] Rulestring [En línia]. LifeWiki (2023). Disponible a: https://conwaylife.com/wiki/Rulestring

[14] tkinter [En línia] (2023). Disponible a: https://docs.python.org/es/3/library/tkinter.html

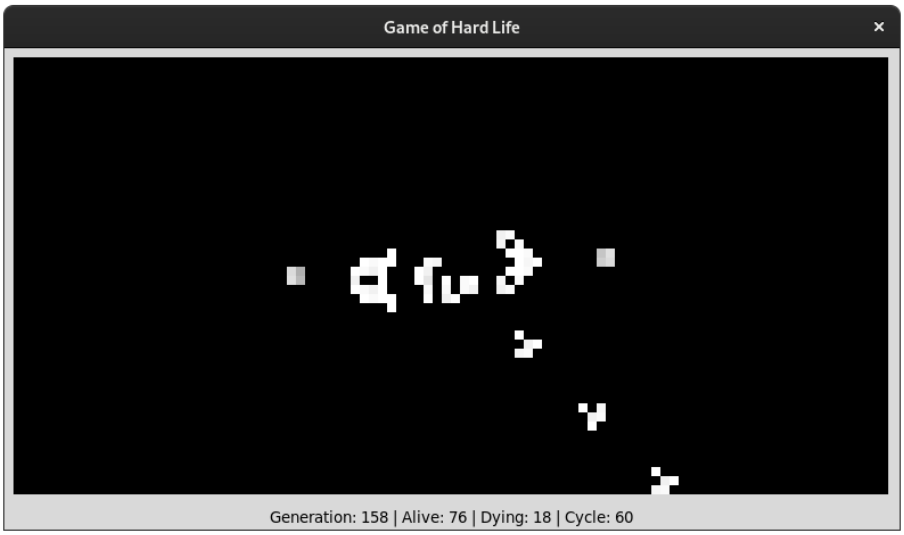


Figura 18: Game of Hard Life - GUI

En aquest entorn, s'hi afegeix la detecció d'ús del teclat sobre les tecles d'espai o de retorn per controlar el flux d'execució: arrancar, aturar o executar una sola generació.

5.2.4 Funcions destacades

`World` :

`randomize()` Aquesta funció es crida per defecte a la inicialització de la classe `World`, i defineix l'estat inicial amb una mostra aleatòria.

```
def randomize(self):
    self.cells = np.random.choice(a=[0, 1], size=self.shape,
                                  p=[0.72, 0.28]).astype(int)
    self.lifecycle = np.multiply(self.cells, self.lifespan)
```

Com és una funció que s'aplica a l'inici, també s'aplica el valor de *lifespan* a cada cel · la.

Encara que és una funció bastant simple, és clau per l'objectiu de trobar comportaments que apareixen de manera natural, en lloc de centrar-se en patrons constructius.

La proporció de pesos més interessant que s'ha descobert de manera general és la que pondera un 72% de cel · les vives.

`update()` Aquesta funció és primordial, ja que estableix l'ordre en què s'executen les diferents passes cada cop que passem a una nova generació.

A l'inici hi ha l'establiment del *timestamp* amb el que controlem el temps de computació que ha suposat calcular els nous estats dels automates cel · lulars.

Seguidament, s'incrementa el comptador de generació i es crida a la funció encarregada d'actualitzar la graella. Un cop fet, es comprova a partir del registre històric si la nova situació coincideix amb algun estat anterior i reconèixer si estem en un estat cíclic.

Finalment, s'enregistra el temps utilitzat, es guarda l'estat actual del món al registre i es retorna l'informe d'estat per actualitzar a la barra informativa.

```
def update(self):
    timestamp_start = time.time()

    ## Update world elements
    self.inc_gen()
    self.cells = self.up_cell()
    self.cycle = self.check_cycles()

    self.uptime = time.time() - timestamp_start

    ## Add new historic registry
    self.registry.reg_hist(self)

    return self.get_stats()
```

Com es pot observar, s'ha tractat de mantenir un estil de codi descriptiu gràcies a l'organització encapsulada de cada procés en diferents funcions. A continuació es descriuen també les funcions `up_cell()`, i `check_cycles()`

`up_cell()` Objectivament la funció més important del programari.

En aquesta funció també es realitza un *timestamp* per reconèixer el temps d'aquesta funció en particular sobre tot el temps invertit dins de l'actualització de la generació.

S'inicialitza la matriu `updated_cells` i es fa un recorregut sobre cada casella de la graella. Per cada una, s'aplica el càlcul de veïns aplicant una estratègia o una altra, depenent de si l'entorn està definit sobre superfície plana o toroidal.

Per defecte s'aplica l'entorn toroidal i es processa el càlcul amb la funció auxiliar `ker(cells, kernel, row, col)`. Aquesta és una de les funcions precompilades per millorar al màxim el temps d'execució.

El *Kernel* és un dels components rellevants que suposen un canvi de paradigma dins d'aquesta variant **Game of Hard Life**, i pot implicar que el resultat del càlcul de veïns no sigui un nombre enter; per això es fa una conversió del resultat del `ker(...)` a nombre enter abans d'analitzar l'estat.

La secció on s'aplica la determinació del nou estat de la cel · la, és un dels components on s'ha aplicat de forma més notable l'estratègia *branchless* descrita a la secció 5.3.1 Estratègies *branchless*.

Aquesta actualització d'estat comprova el valor amb les condicions que han vingut definides a partir de la *Rulestring*, i apliquen l'estat de viu o mort a la casella tenint en compte també el component de temps màxim de vida, que és precisament el més rellevant de la varietat **Game of Hard Life**.

Un cop s'ha recorregut i actualitzat tota la graella, es fa una comparativa de cel · les mortes en aquesta generació per incorporar al conjunt de dades informatives de l'estat, i tot seguit es tanca el comptador de temps i es retorna el valor de la matriu de cel · les actualitzada.

```
def up_cell(self):

    timestamp_start = time.time()
    self.updated_cells = np.zeros(self.shape, dtype=int)

    for row, col in np.ndindex(self.shape):
        if self.toroidal:  ## TOROIDAL
            neighbours = ker(self.cells, self.kernel, row, col)

        else:  ## FLAT
            neighbours = np.sum(self.cells[row - 1:row + 2, col - 1:col + 2])
                                - self.cells[row, col]

        neigh = str(int(np.trunc(neighbours)))
        if (self.cells[row, col] > 0 and (neigh in self.survival)) \
                or (self.cells[row, col] == 0 \
                    and (neigh in self.birth) \
                    and self.respawn((row, col))):
            self.updated_cells[row, col] = int(
                1 * ((HARD and (self.lifecycle[row, col] > 0)) or not HARD))

    self.dying = dying_cells(self.cells, self.updated_cells)
    self.ctime = time.time() - timestamp_start
    return self.updated_cells
```

El refinament dins aquest bloc de codi ha estat el punt substancialment més complex de tot el programari desenvolupat fins a aquesta etapa, i el que ha anat evolucionant durant les diferents fases del projecte.

`check_cycles()` Aquesta funció utilitzada també dins del procés de `update()` és la que revisa dins el registre històric si es dona una situació cíclica.

El registre està limitat a una llista de còpies de les graelles anteriors fins a un cert valor límit que es configura amb les variables d'entorn, que pren per defecte el doble del valor de temps de vida que tenen les cel · les.

Aquesta llista va incorporant les matrius de cel · les segons el nombre de generació modulat per la mida màxima del registre. D'aquesta manera, i

descrit al mètode `hist_pos()` de la classe `Registry`, en qualsevol moment podem saber sobre quina posició de l'històric es correspon la generació actual.

Un cop obtinguda aquesta posició, es recorre cada generació anterior aplicant una distància modular, ja que d'aquesta manera podem fer un recorregut cíclic per tot el llistat indistintament de quin és el punt on l'hem iniciat, anant fins a l'element inicial de la llista i continuant des del darrer, fins a acabar tot el cicle a la posició que correspon a aquesta generació.

Si s'ha donat alguna coincidència amb alguna de les matrius de l'històric, es retorna aquesta distància modular. En canvi, si no s'ha trobat cap coincidència, es retorna el valor cero.

```
def check_cycles(self):
    pos = self.registry.hist_pos(gen=self.gen)
    for n in range(1, len(self.registry.history)):
        ## Check history regs backwards
        i = (pos - n) % self.registry.hsize
        if (self.cells == self.registry.history[i]).all():
            ## Modular distance in cyclic list
            return (pos - i) % self.registry.hsize

    return 0
```

Global :

`ker(cells, kernel, row, col)` Aquesta funció és de major rellevància, no només pel seu paper a l'hora de fer el càlcul essencial a partir de l'entorn, sinó també pel fet que s'executa dins el bloc principal que haurà recorregut cada cel · la. Per tant, cada operació que s'aplica dins d'aquesta funció s'executa $n \times m$ cops cada generació, essent n, m les dimensions de la graella.

El marcador `@njit` a sobre de la definició, és l'indicador amb el qual s'aplica el precompilat previ, precisament degut a l'alt ús reiteratiu d'aquesta funció, i que s'explica a la secció 5.3.2 Precompilat.

En aquesta funció entra en joc el vector de definició del *Kernel*, del que hem de conèixer la grandària, ja que el nombre d'elements d'aquest vector comporta implícitament el rang de distància segons la regla de Moore. Això és un valor que pot variar segons la configuració de l'entorn i no podem pressuposar res, per això el primer pas és determinar el valor `ksize`.

Seguidament, es recorren les posicions dins del rang del *Kernel* i, per cada veí, s'aplica el coeficient de pès que té segons la seva distància.

Hem de recordar que el càlcul de la posició dins de la graella `cells` és modular, ja que s'aplica la geometria cílica implícita del model de superfície toroidal.

Finalment, es retorna el valor del càlcul dels veïns considerats d'acord amb els coeficients de pesos definits segons el *Kernel*.

```
@njit#(parallel=True)
def ker(cells, kernel, row, col):
    ksize = kernel.shape[0]
    neighbours = 0
    for i, j in [(x, y) for x in prange(-ksize + 1, ksize) for y in prange(-ksize + 1, ksize)]:
        neighbours += np.multiply(
            cells[(row + i) % X, (col + j) % Y],
            kernel[np.maximum(np.abs(i), np.abs(j))]
        )

    return neighbours
```

5.3 Refinament

5.3.1 Estratègies *branchless*

La metodologia *branchless* és una estratègia a l'hora de generar codi que tracta d'optimitzar el nombre d'operacions que realitza una certa funcionalitat.

El plantejament es basa en el fet que les operacions de tipus *if* o *switch* impliquen, a l'hora de traduir el codi a llenguatge màquina del processador, la generació de salts entre blocs d'ordres que poden implicar també carregar cada cop nous conjunts de dades a la CPU segons el cas. Però aplicant una estratègia *branchless* es pot resoldre convertint els condicionals en una combinació d'operacions lògiques o factors multiplicatius que donen el mateix resultat.

Un exemple simple seria convertir el següent condicional:

```
if a > b:
    return a
else:
    return b
```

en una simple operació que combina factors multiplicatius sobre resultats lògics:

```
return (a > b) * a + (a <= b) * b
```

Com es pot observar, el segon resultat és molt menys intuïtiu que el primer a l'hora de fer-ne lectura, però evita que el processador hagi de carregar un bloc d'ordres o un altre depenent de la condició `(a < b)`, si no que multiplica el valor d'aquesta operació lògica pel valor de la variable; en ambdós casos, un dels components es multiplica per zero, que se suma al valor de l'altre variable donant així el mateix resultat.

La part positiva evidentment és trobar una solució de codi que pot oferir la mateixa solució amb un cost computacional menor, i per això millora el rendiment operatiu de la funcionalitat.

La contra és que el bloc de codi normalment no és massa evident a l'hora de fer-ne lectura, i fa que sigui particularment complicat entendre què està fent o com s'ha de modificar si necessitem intervenir en el seu funcionament, ja sigui per afegir nous components a tenir en compte o per fer manteniment i arreglar possibles casos d'error.

Precisament per aquest component críptic que deixa al codi, la metodologia *branchless* s'aplica normalment a funcions molt concretes que es coneix que s'han d'executar repetidament molts de cops, i el seu impacte és realment rellevant.

El bloc on s'ha utilitzat de forma més notable és dins la funció `up_cells()`:

```
        if (self.cells[row, col] > 0 and (neigh in self.survival)) \
                or (self.cells[row, col] == 0 \
                    and (neigh in self.birth) \
                    and self.respawn((row, col))):
            self.updated_cells[row, col] = int(
                1 * ((HARD and (self.lifecycle[row, col] > 0)) or not HARD))
```

Primerament, s'ha simplificat en un sol condicional *if* el que es podria haver interpretat com a bloc de sub-condiciconals:

```
if (self.cells[row, col] > 0 and (neigh in self.survival)):
    self.updated_cells[row, col] = ## ...
elif (self.cells[row, col] == 0):
    if (neigh in self.birth):
        self.respawn((row, col)
        self.updated_cells[row, col] = #...
```

El segon sembla una mica més explicatiu, però suposa una ramificació de blocs d'ordres que acaben a la CPU.

L'altre punt d'aplicació és sobre l'ordre d'actualització de l'element de la graella, que es mostra com:

```
self.updated_cells[row, col] = int(
    1 * ((HARD and (self.lifecycle[row, col] > 0)) or not HARD))
```

Aquesta línia sembla una mica críptica, això no obstant, el rendiment resulta més òptim que no pas si s'haguera escrit de forma més descriptiva, com per exemple:

```
if HARD:
    if self.lifecycle[row, col] > 0:
        self.updated_cells[row, col] = 1
    else:
        self.updated_cells[row, col] = 0
else:
    self.updated_cells[row, col] = 1
```

Per les proves que s'han incorregut durant les fases de desenvolupament, l'aplicació d'aquesta estratègia ha suposat una millora reduint el temps total de la funció `update()` aproximadament en un factor 100 cops més eficient, usant una graella $X = 192, Y = 96$.

5.3.2 Precompilat

Python és un llenguatge de programació d'alt nivell i interpretat que utilitza tipat dinàmic per la gestió de memòria. Aquestes característiques atorguen un gran dinamisme, i en part és una de les raons de la seva popularitat i del seu ís estès dins de la branca de ciència de dades, però implica també que el temps d'execució no és molt eficaç.

Això es nota particularment a l'hora de tractar recorreguts de matrius grans i en cert punt de la segona fase de desenvolupament, els temps d'execució consumits per a calcular un pas de generació arribaven a ser inacceptables.

Malgrat això, un gran avantatge addicional és la disposició de llibreries que treballen fent servir funcions precompilades en llenguatges més eficients com són *C/C++*, com és el cas de *numpy*.

A més, s'ha emprat la llibreria *numba* que ens permet definir sobre una funció l'indicador de precompilat, de manera que el primer cop que es fa una crida a aquesta funció, es genera una versió precompilada que queda en memòria,

i les pròximes execucions d'aquesta mateixa funció durant la resta del temps d'execució són resultat d'aquesta versió compilada.

L'ús d'aquesta eina suposa una sèrie de condicions, com la necessitat de definir la funció dins l'àmbit global i no com mètode dins d'una classe, o que les dades que s'han de tractar dins aquesta funció han de ser de tipus primari o accepta també matrius de *numpy*, però no podrà reconèixer estructures de dades pròpies o referències a objectes.

És per aquesta raó que les funcions `ker()` i `dying_cells()` es troben a l'àmbit global, i no poden tenir referències a propietats com `world.cells`: s'han de passar específicament les dades a treballar i recollir el resultat.

Tot i aquestes limitacions, el resultat ha estat molt notable, ja que els temps de computació s'han reduït en dos o tres ordres de magnitud.

És cert que la primera execució resulta una mica més lenta que si no fem servir *numba*, però això és degut al temps que internament prepara la versió precompilada de la funció. Així i tot, és un cost que s'amortitza des de la següent execució indiscutiblement, i en totes les generacions posteriors.

Aquest paquet és particularment útil per definir sobre funcions que s'executen molts cops i que fan operacions sobre elements bàsics.

Sobre una funció marcada amb aquesta llibreria, genera una pre-compilació optimitzada en C, de manera que Python no ha de fer el seu procés intern de convertir les línies del *script*.

Precompilar la funció clau de càlcul de veïns entorn de cada cella, ha reduït el temps de computació entre dos i tres ordres de magnitud.

5.3.3 Definició de l'estructura de dades de sortida

L'execució de cada simulació genera un fitxer amb informació de la configuració de l'entorn i del registre evolutiu de l'estat del món a cada generació. Aquest fitxer es genera a partir de la classe `Registry`, que s'utilitza amb la definició de l'objecte de tipus `World`, i tant és un producte obtingut tant si s'executa amb la interfície gràfica d'usuari, com si s'executa des de l'agent en un terminal.

El format de sortida escollit és YAML, ja que és un standard prou estès i reconegut, que és supergrup del format JSON, i aporta millor facilitat de lectura humana mantenint les propietats estructurals.

Els camps globals de cada experiment són:

`hsize` Mesura del llistat on s'enregistra l'històric de l'estat de la matriu de cel · les

`inf_loop` Valor booleà, marca si en trobar-se un estat cíclic comporta l'aturada del flux de simulació

`lifespan` Temps de vida màxim que pot mantenir una cel · la de forma continuada

`sape` Dimensions de la graella

`stats` Llistat de dades de l'estat del món per a cada generació que inclou:

`alive` Nombre de cel · les vives

`ctime` Temps de computació de la funció `up_cell()`

`cycle` Indicador d'estat cíclic

`dying` Nombre de cel · les que moren en aquesta generació

`gen` Nombre de la generació

`kernel` Valors del vector *Kernel*

`rulestring` La *Rulestring* aplicada en aquest experiment

`uptime` Temps de computació de la funció `update()`

`toroidal` Valor booleà, marca si s'aplica la geometria toroidal

Aquestes dades han estat plantejades per poder analitzar principalment la proporció d'automates cel · lulars vius i la previsió de quants van a morir, especialment per comparar com afecten diferents valors de les variacions del *Kernel* o de la *Rulestring* a les següents generacions.

Els fitxers generats són per tant una eina analítica que, si bé es pot llegir de forma humana, la intenció final és que essent una col · lecció de dades estructurada, serveixi per l'anàlisi automatitzat per part de l'agent.

5.4 Simulacres

En aquesta secció es mostren resultats de proves fetes per pròpia exploració i construcció dins la secció 5.4.1 Exemples en GUI, i les proves fetes a partir de les dades analitzades a l'apartat 5.1.4 A-8, Anàlisi sobre l'entorn producte: *Game of Hard Life - 1.0*

5.4.1 Exemples en GUI

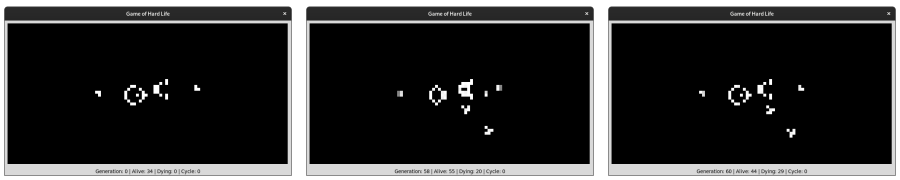
Glider Gun

Un primer cas ha estat tractar de replicar la figura *Glider Gun* del propi *Conway's Game of Life* original. Per això s'ha aplicat la configuració inicial `init_form = forms.GliderGun` al fitxer `settings.py` però mantenint les condicions de la varietat que estem tractant.

Una condició molt important ha estat fer coincidir el període de caducitat amb el període de la figura, ja que si no, en algun moment quan una de les formes necessita utilitzar els blocs de pivot, si aquest es troba en el seu moment de reciclatge, la figura perd la seva forma i acaba col · lapsant.

És una mostra de que en aquesta varietat, moltes de les figures reconegudes i catalogades no són estables en aquest cas, o poden necessitar una revisió de condicions per a segui essent viable en aquest entorn, perquè la vida és dura.

A la següent figura es mostra l'evolució en els primers períodes de sincronia en que es van produint els nous *gliders*:



(a) Game Of Hard Life (1.0): [gen:0] Glider Gun (b) Game Of Hard Life (1.0): [gen:58] Glider Gun (c) Game Of Hard Life (1.0): [gen:60] Glider Gun

Figura 19: Game of Hard Life (GUI): *Glider Gun*

Rosa del desert

Aquesta és una configuració descoberta per simple exploració modificant les condicions, i se l'hi ha donat aquest nom per la similitud del resultat amb el seu mineral homònim.

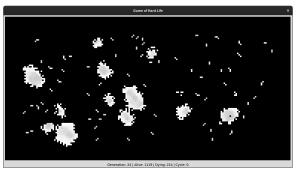
En aquesta s'utilitza una *Rulestring* de caràcter expansiu: `B3567/S15678`, mantenint el *Kernel* normal.

Segueix un comportament de tipus *invasiu* com s'explica al següent apartat, on les primeres generacions venen marcades per una gran purga de cel · les mentre col · lapsen en uns pocs nuclis estables de població.

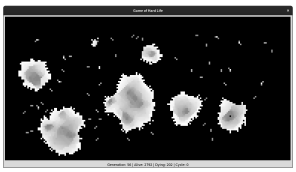
Aquests nuclis van creixent de forma expansiva i son bastant tolerants a la caducitat de cada cel · lula, ja que formen una superfície d'expansió que pot anar reformant aquelles que es van perdent des de l'interior d'aquests nuclis de població.

La propietat més interessant d'aquesta configuració és que aquests van cobrint de forma bastant ordenada el mapa, i poden arribar a un estat cíclic que assegura la seva permanència.

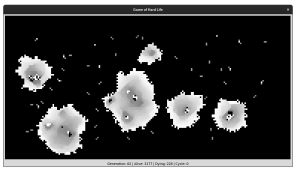
(a) Game Of Hard Life (1.0): [gen:8] Rosa del desert



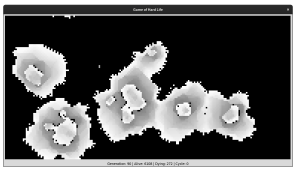
(b) Game Of Hard Life (1.0): [gen:24] Rosa del desert



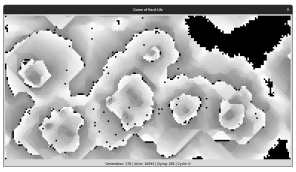
(c) Game Of Hard Life (1.0): [gen:56] Rosa del desert



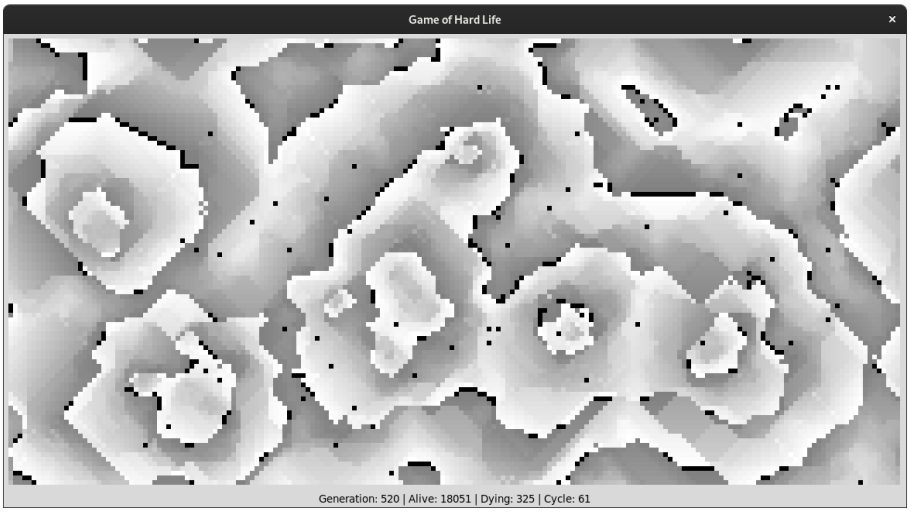
(d) Game Of Hard Life (1.0): [gen:62] Rosa del desert



(e) Game Of Hard Life (1.0): [gen:90] Rosa del desert



(f) Game Of Hard Life (1.0): [gen:178] Rosa del desert



(g) Game Of Hard Life (1.0): [gen:520] Rosa del desert

Figura 20: Game of Hard Life (GUI): *Rosa del desert*

5.4.2 Exemples Agent

Seguint les mostres presentades a la figura A-8, Anàlisi sobre l'entorn producte: *Game of Hard Life - 1.0*, s'han escollit els tests elaborats a partir de l'agent utilitzant particions del vector *Kernel* en cinc per mostrar a la memòria. Hi ha més tests recollits al corresponent fitxer adjunt.

Com es pot apreciar a les gràfiques, hi ha almenys dos línies de comportament generals:

polse Un inici de creixement expansiu fins que cobreix una certa cota, i que un cop arribat segueix un patró serrat, que segons avancen les generacions queda cada cop més acotat. El període que marca les caigudes a la gràfica provocant aquest efecte de serra, ve marcat pel valor del temps maxim de vida.

invasiu Es perd des d'un inici gran part de la població, formant petits focus de població, però que lentament van cobrint el mapa i es mantenen amb relativa estabilitat. No s'observa un efecte tan marcat del període del temps de vida.

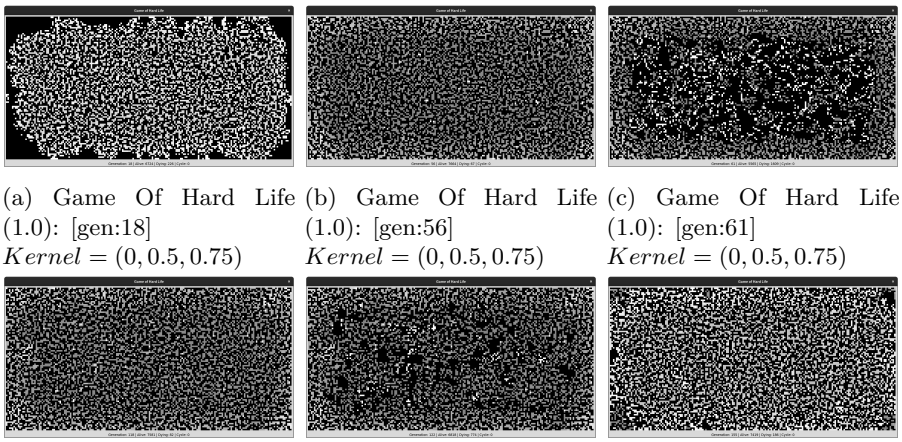
A la següent figura, es mostra el resultat obtingut amb el GUI configurant els valors de *Kernel* i *Rulestring* segons la prova en concret de les que reprodueixen el comportament de tipus *polse*. Concretament la que correspon a la figura A-8, Anàlisi sobre l'entorn producte: *Game of Hard Life - 1.0* amb $Kernel = (0, 0.5, 0.75)$.

Podem veure a les captures com en les primeres generacions s'ha cobert pràcticament la totalitat de la superfície, però moltes d'aquestes cel · les queden fixes, i per tant a les generacions properes a la vida màxima, es troben gris degut al degradat que mostra com es deteriora amb el temps. Un cop hi passa el període crític, es veu que hi ha una regeneració que torna a formar-se a partir dels nous espais i es repeteix l'expansió.

Aquest procés es va repetint, quedant aquests espais cada cop més disseminats per tota la graella, i el període cada cop va afectant menys a la renovació de la població.

Tot això explica prou bé el que veiem a la gràfica, i tot i que no mostra gran interés pel que fa a les formes, és un comportament emergent que sorgeix de manera global.

En general, s'ha vist que aquesta és la forma d'actuar amb regles que propicien una ràpida expansió de cel · les; apareix de forma natural la tendència al comportament de tipus *polse*.



(a) Game Of Hard Life (1.0): [gen:18] $Kernel = (0, 0.5, 0.75)$

(b) Game Of Hard Life (1.0): [gen:56] $Kernel = (0, 0.5, 0.75)$

(c) Game Of Hard Life (1.0): [gen:61] $Kernel = (0, 0.5, 0.75)$

(d) Game Of Hard Life (1.0): [gen:118] $Kernel = (0, 0.5, 0.75)$

(e) Game Of Hard Life (1.0): [gen:122] $Kernel = (0, 0.5, 0.75)$

(f) Game Of Hard Life (1.0): [gen:155] $Kernel = (0, 0.5, 0.75)$

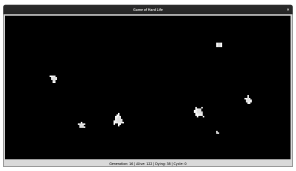
Figura 21: Experiments A07 - Game of Hard Life (GUI): *polse*

Per acabar, a la pròxima figura, es mostra el resultat obtingut amb el GUI configurant els valors de *Kernel* i *Rulestring* segons la prova en concret de les que reprodueixen el comportament de tipus *invasiu*. Concretament la que correspon a la figura A-8, Anàlisi sobre l'entorn producte: *Game of Hard Life - 1.0* amb $Kernel = (0, 1, 0)$.

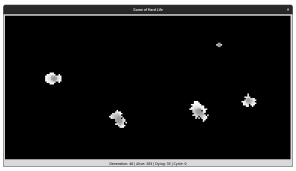
Podria dir-se que aquesta configuració és poc tolerant a l'estat aleatori, i en les primeres generacions gran part de la població mor ràpidament, quedant uns pocs nuclis on col · lapsen alguns grups de cel · les que formen el que se'n podria dir uns brots inicials. Aquests brots van creixent de forma lenta mentre formen una superfície variant, i això fa que siguin bastant resistents a l'efecte del període màxim de vida.

Por a poc, aquestes construccions van cobrint el món de manera invasiva. Fins ara, aquest comportament és el que s'assembla més a un patró de creixement que puguem anomenar com a més organic

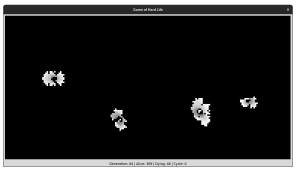
(a) Game Of Hard Life (1.0): [gen:8] $Kernel = (0, 1, 0)$



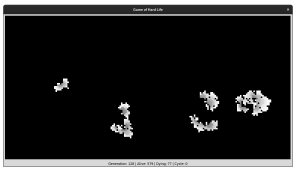
(b) Game Of Hard Life (1.0): [gen:16] $Kernel = (0, 1, 0)$



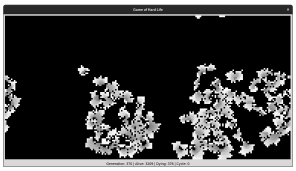
(c) Game Of Hard Life (1.0): [gen:48] $Kernel = (0, 1, 0)$



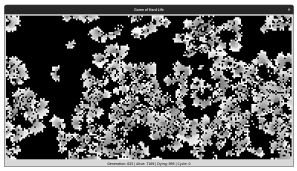
(d) Game Of Hard Life (1.0): [gen:64] $Kernel = (0, 1, 0)$



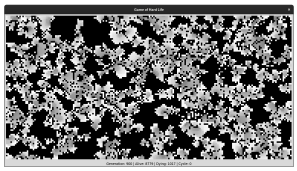
(e) Game Of Hard Life (1.0): [gen:128] $Kernel = (0, 1, 0)$



(f) Game Of Hard Life (1.0): [gen:370] $Kernel = (0, 1, 0)$



(g) Game Of Hard Life (1.0): [gen:625] $Kernel = (0, 1, 0)$



(h) Game Of Hard Life (1.0): [gen:900] $Kernel = (0, 1, 0)$

Figura 22: Experiments A07 - Game of Hard Life (GUI): *invasiu*

6 Discussió

Al llarg d'aquest treball, s'ha aconseguit tenir una primera versió funcional de la varietat *Game of Hard Life*, i s'ha pogut construir el primer prototip del procés agent que s'encarrega de realitzar proves amb diferents configuracions exportant les metadades del registre de la simulació.

Si bé s'han trobat diversos obstacles al llarg del camí, s'ha pogut trobar una solució raonable encara que hagi estat a cost d'invertir un major temps, però oferint a aquestes primeres versions inicials de major solidesa. Malauradament, no s'han pogut acabar alguns dels objectius opcionals del procés agent, i que queden pendents com es mostra a l'apartat de 7.2 Línies de futur.

Resta dir que és relativament simple implementar una versió mínima com exercici a petita escala; de fet, existeixen molts exemples en línia d'implementacions bàsiques com exercici de programació inicial, però quan s'hi aprofundeix, sorgeixen molts paradigmes possibles que n'augmenten la complexitat i és quan es posa a prova la capacitat d'abstracció i la recerca de solucions.

En aquest sentit, el fet de descobrir, practicar i aprendre les tècniques esmentades durant el treball de precompilat de funcions o de les estratègies *branchless*, és una mostra de l'aplicació de solucions per fer front a les dificultats que han anat sorgint, sobretot en l'àmbit que ha sorgit d'optimització dels processos interns del programa.

Actualment, existeixen diversos algorismes que poden accelerar aquests processos amb el simulador original, però no han estat fàcilment extrapolables perquè el reconeixement de patrons no funciona de la mateixa manera en aquesta varietat, ja que la caducitat dels automates cel · lulars implica una major complexitat al model que suposa una revisió important d'aquestes implementacions.

Després de 50 anys, hi ha molta investigació acumulada sobre l'original *Conway's Game of Life* i s'han dissenyat nombroses varietats que estenen el model tant en l'àmbit teòric, com en l'elaboració de patrons enginyosos i de gran complexitat. Així i tot, amb aquest treball es mostra que tot i aquest històric d'investigació i desenvolupament, encara poden presentar-se noves condicions i reinterpretacions.

Després de tot, tot i les diferents millores que han quedat pendents, l'objectiu inicial d'implementar la fase inicial de la varietat *Game of Hard Life* ha obtingut un resultat funcional, i s'ha pogut utilitzar el procés d'agent per investigar i descobrir alguns patrons d'interès, tot i que no es tracta de figures catalogables de forma determinística, sinó de comportaments emergents que es donen observant a tota la població com a conjunt, com han estat els casos descrits de comportaments tipus *polse* o tipus *invasiu*.

Finalment, és prou interessant haver descobert el comportament descrit al final de la secció de resultats Exemples Agent amb la figura de `Experiments A07 - Game of Hard Life (GUI):` *invasiu*.

7 Conclusions

7.1 Conclusions

En aquest treball s'han assolit els objectius principals pel que fa a l'elaboració d'un programa funcional que reprodueix el model de la varietat *Game of Hard Life*, i s'han pogut analitzar alguns dels primers comportaments emergents que han sorgit durant l'experimentació.

En aquest sentit, el treball ha complit amb l'objectiu central. Tot i així, hi ha molts components addicionals que encara es poden continuar treballant, tant pel que fa al propi projecte, com per l'adaptació de tantes altres possibilitats que ja compta el simulador original.

Addicionalment, ha servit per descobrir pràctiques i estratègies d'optimització que han resultat molt interessants i enriquidores, que de per sí són extrapolables a pràcticament qualsevol àmbit de la programació; per tant, encara que hagi sorgit a partir d'un experiment més bé recreatiu, les lliçons apreses han estat prou interessants per fer servir en molts altres escenaris.

Conway's Game of Life és en sí mateix un projecte que acumula una llarga història de desenvolupament i investigació, i difícilment es poden cobrir totes les possibilitats elaborades fins ara en un sol treball; però si aquesta varietat produeix major interès, per molta d'aquesta feina feta només cal analitzar de quina manera es pot aplicar una estratègia general d'adaptació i que passi a formar part com una variació més del seu gran ecosistema.

7.2 Línies de futur

Aquest treball encara pot optar a afegir noves capacitats i completar-se amb funcionalitats afegides.

Alguns objectius opcionals o millores addicionals poden ser:

- Model de reconeixement de figures notables sobre el mapa, tenint en compte les condicions pròpies de caducitat.
- Afegir a l'agent les capacitats de reconeixement de figures en temps real.
- Millorar la gestió de processos en paral · lel.
- Aplicació de l'algorisme *HashLife* adaptat a les condicions de la varietat.
- Millorar l'experiència i control de l'usuari (UX) a l'aplicació amb GUI.
- Afegir a l'agent capacitat analítica en temps d'execució de les dades del registre.
- Obtenir una xarxa neural de pesos corresponents als *Kernel* òptims segons la situació d'una generació.

7.3 Seguiment de la planificació

A l'annexa de *Full de seguiment* hi ha la descripció detallada de cada tasca, i es determinen les respectives valoracions i comentaris.

Respecta al seguiment de la planificació, principalment es detallen les tasques que han estat relacionades en l'esforç d'optimització del codi i alguns canvis pel que fa a la presentació i estructuració de dades experimentals.

Al següent apartat es dona la descripció general del seguiment.

7.4 Grau de compliment dels objectius i resultats previstos en el pla de treball.

Codi	Objectiu	Requisit	Data límit
✓ I-01	Definició inicial del projecte	∅	13/03
✓ I-02	Pla de treball	∅	28/03
	Fase I		
✓ A-01	Recopilació de dades experimentals a partir de l'entorn base *Game of Life*	∅	03/04
∅ A-02	Anàlisi de les dades experimentals A-01	A-01	07/04
✓ G-01	Preparació de l'entorn de desenvolupament	∅	07/04
✓ G-02	Desenvolupament de l'entorn inicial *Game of Life*	G-01	11/04
✓ G-03	Versió inicial de *Game of Hard Life*	G-02	14/04
✓ A-03	Recopilació de dades experimentals a partir de l'entorn *Game of Hard Life - betta*	G-03	17/04
∅ A-04	Anàlisi de les dades experimentals A-03	A-03	19/04
✓ G-04	Varietat: activació d'entorn toroidal	G-03	26/04
✗ G-05	Varietat: Configuració de condicions de vida o mort	G-03	26/04
✓ G-06	Varietat: Configuració de la caducitat de les caselles	G-03	26/04
✓ A-05	Recopilació de dades experimentals a partir de l'entorn *Game of Hard Life - 0.3*	G-06	28/04
∅ A-06	Anàlisi de les dades experimentals A-05	A-05	03/05
∅ D-01	Memòria del TFG (Fase 1)	∅	03/05
	Fase II		
✓ G-05 (revisió)	Varietat: Configuració de condicions de vida o mort	G-03	26/04
✓G-07	API d'obtenció de metadades	G-06	17/05
✓G-08	API de control d'entorn	G-06	17/05
∅G-09	Agent de control de població	G-07/8	24/05
✓A-07	Recopilació de dades experimentals a partir de l'entorn *Game of Hard Life - 1.0*	G-09	26/05
✓A-08	Anàlisi de les dades experimentals A-07	A-07	29/05
✓D-02	Memòria del TFG (Fase 2)	D-01	29/05
	Fase Final		
✓A-09	Anàlisi de comportaments emergents	A-08	20/06
✓D-3	Memòria del TFG (**Final)**	D-02	20/06

Taula 2: Seguiment d'objectius

7.5 Justificació dels canvis en cas necessari

Interfície

Inicialment, es tenia plantejat utilitzar el paquet o *framework* per *Python*: *PyGame* https://www.pygame.org.

Però s'ha vist que és un paquet orientat a l'experiència de jugar, i encara que el *Game of Life* sembla que pot encaixar, en realitat no necessitem l'entorn d'interacció, ja que aquest és un *zero-player*. D'altra banda, també que ha semblat convenient cercar una opció d'entorn més generalista per mostrar els resultats analítics amb gràfiques a la Fase II.

A mesura que ha avançat el projecte, s'ha decidit reformular-ho per la paqueteria de la interfície *tkinter* (https://docs.python.org/es/3/library/tkinter.html). A més, aquesta mostra un desenvolupament y una documentació molt més rigurosa i professional.

Detecció de l'estat cíclic

La realització dels experiments sobre l'entorn inicial de *Game of Hard life* ha implicat la revisió de les funcions destinades a detectar l'estat cíclic, i no es descarta que es tingui que seguir millorant més endavant, degut a la complexitat afegida per les noves condicions.

Memòria en format LaTeX

S'ha adaptat la plantilla de documentació -la memòria del TFG- a format LaTeX.

Aquest format és referent dins l'àmbit científic i acadèmic, i així com aquest treball porta una part tècnica, més endavant s'espera guanyar pés en la part analítica a l'hora d'explicar els conceptes de caràcter teòric i matemàtic que s'empraran cap al final de la Fase II.

Stats

El registre d'estats mostrava inicialment sobre el terminal un llistat que representant l'històric de cel · les vives, però aquesta informació és insuficient.

Per tal de millorar les possibilitats analítiques, s'ha estructurat com a diccionari incorporant tota la informació d'estat que arriba de la funció del model *World*, i s'ha afegit la funcionalitat de guardar aquesta informació en fitxers YAML.

Això permetrà major capacitat de decisió a l'agent quan corre els seus simulacres, i també aporta la possibilitat de realitzar processos de minería de dades.

Optimització del Software

A mesura que ha anat guanyant complexitat, les rutines de generació dels estats següents suposaven cada cop més temps de computació.

És un fet reconegut que el llenguatge Python, a causa de la seva major capa d'abstracció i la falta de rigidesa en les estructures de dades i tipus de variables utilitzats, no és massa òptim en el tractament de dades grans.

En un moment donat, el fet d'afegir una sola operació més al procés d'obtenir l'estat de l'entorn de cada cel · la, els temps es multiplicaven en diversos ordres de magnitud, fent que cada generació suposi més d'un segon i resulti impracticable fer un nombre extens de proves.

Per contrarestar aquest efecte, s'han aplicat dues estratègies de mitigació per millorar la *performance* del codi:

Branchless L'aplicació d'estratègies *branchless* tracten de reinterpretar situacions en les quals el codi genera ramificació, com ocorre amb clàusules tipus *if* o *switch*, on el processador ha d'estructurar les ordres amb salts i carrecar nous conjunts d'ordres segons el cas del condicional, per operacions que combinen operadors lògics que acaben donant el mateix resultat, però amb un conjunt fixat i normalment reduït d'ordres quan es compila al processador.

Aquesta tècnica pot millorar el rendiment d'apartats de codi que s'han d'executar molts cops. Per contra, el codi resultant és menys intuïtiu.

Els detalls de l'aplicació de l'estratègia de programació *branchless* es donen a la memòria del treball.

Numba.njit Aquest paquet és particularment útil per definir sobre funcions que s'executen molts cops i que fan operacions sobre elements bàsics.

Sobre una funció marcada amb aquesta llibreria, genera una pre-compilació optimitzada en C, de manera que Python no ha de fer el seu procés intern de convertir les línies del *script*.

Precompilar la funció clau de càlcul de veïns entorn de cada cella, ha reduït el temps de computació entre dos i tres ordres de magnitud.

8 Glossari

B3/S23 La *Rulestring* corresponent al joc original de *Conway's Game of Life*

Branchless Estratègia a l'hora de generar codi que tracta d'optimitzar el nombre d'operacions executades a la CPU que realitza una certa funcionalitat, que es centra en transformar l'ús de codi condicionat per operacions de tipus lògica o aritmètica amb el mateix resultat

Conway's Game of Life Joc dissenyat originalment per *John Conway* a l'any 1970 i que s'ha seguit desenvolupant, i que és la base sobre la que es desenvolupa la variant d'aquest treball

CPU Unitat central de processament (*Central Processing Unit*)

Glider A *Conway's Game of Life*, patrons o figures que segueixen una evolució estable acompanyada d'un desplaçament durant el pas de les generacions

GUI Interfície gràfica d'usuari (*Graphical User Interface*)

HashLife (algorisme)

Kernel Vector que representa la consideració de pesos segons la distància de Moore a tenir en compte per cada cel · la pel que fa a la comptabilització dels veïns del seu entorn a l'hora d'analitzar el seu proper estat segons les condicions de despoblació o sobrepoblació

Oscilator A *Conway's Game of Life*, patrons o figures que van variant segons un cert període de manera estable durant el pas de les generacions

Python Llenguatge de programació d'alt nivell i interpretat que utilitza tipat dinàmic per la gestió de memòria

Rulestring Standard de notació en format *string* amb la qual es defineix el comportament dels automates cel · lulars segons la informació obtinguda del seu entorn

Still life A *Conway's Game of Life*, patrons o figures que es mantenen estables durant el pas de les generacions

Toroide Superfície de revolució generada per un polígon o curva plana tancada simple, que gira al voltant d'una recta exterior coplanar com eix de rotació amb la qual no interseca. En aquest treball es tracta el cas del toroide generat per un cercle amb radi de rotació major al radi del cercle generador, conegut concretament com a toro d'anell

9 Bibliografia

- Mòduls

  Presentació de documents i elaboració de presentacions (HKZB5UG6XS130_6R5O43)

  Redacció de textos cientificotècnics (8QROP4G6IXT6ND3J1_XE)

  dels materials de l'assignatura **TFG - Intel.ligencia Artificial**
- *Creative Commons — Attribution-NonCommercial-ShareAlike 4.0* International — CC BY-NC-SA 4.0 [En línia] Creative Commons [consulta: juny 2023]. Disponible a: https://creativecommons.org/licenses/by-nc-sa/4.0/
- *Conway's Game of Life* [En línia]. Wikipedia [consulta: juny 2023]. Disponible a: https://en.wikipedia.org/wiki/Conway's_Game_of_Life
- *LifeWiki* [En línia]. LifeWiki [consulta: juny 2023]. Disponible a: https://conwaylife.com/wiki

  List of Life-like rules [En línia]. LifeWiki [consulta: juny 2023]. Disponible a: https://conwaylife.com/wiki/List_of_Life-like_rules

  List of common still lifes [En línia]. LifeWiki [consulta: juny 2023]. Disponible a: https://conwaylife.com/wiki/List_of_common_still_lifes

  Moore neighbourhood [En línia]. LifeWiki [consulta: juny 2023]. Disponible a: https://conwaylife.com/wiki/Moore_neighbourhood

  Rulestring [En línia]. LifeWiki [consulta: juny 2023]. Disponible a: https://conwaylife.com/wiki/Rulestring
- *GitLab* [En línia] [consulta: juny 2023]. Disponible a: https://gitlab.com
- *Life Viewer* [En línia] [consulta: juny 2023]. Disponible a: https://lazyslug.com/lifeviewer
- *Golly Game of Life Home Page* [En línia] [consulta: juny 2023]. Disponible a: https://golly.sourceforge.net
- *John Conway's Game of Life - An Introduction to celluar Automata* [En línia] [consulta: juny 2023]. Disponible a: https://beltoforion.de/en/game_of_life
- *Catalogue* [En línia] [consulta: juny 2023]. Disponible a: https://catagolue.hatsya.com/home
- *Conway's Game of Life on a Torus - Tim Hutton* [En línia]. Youtube.com (2023). Disponible a: https://www.youtube.com/watch?v=lxIeaotWIks
- *tkinter* [En línia] [consulta: juny 2023]. Disponible a: https://docs.python.org/es/3/library/tkinter.html
- *PyGame* [En línia] [consulta: juny 2023]. Disponible a: https://www.pygame.org
- *5. Data Structures — Python 3.11.3 documentation* [En línia]. python.org [consulta: juny 2023]. Disponible a: https://docs.python.org/3/tutorial/datastructures.html
- *Numba documentation — Numba 0.50.1 documentation* [En línia]. Numba.org [consulta: juny 2023]. Disponible a: https://numba.pydata.org/numba-doc/latest/index.html
- *PyYAML Documentation* [En línia]. PyYAML.org [consulta: juny 2023]. Disponible a: https://pyyaml.org/wiki/PyYAMLDocumentation
- JOHNSTON, Nathaniel, GREENE, Dave. *Conway's Game of Life - Mathematics and Construction* [En línia]. Autopublicat [2022]. ISBN 978-1-794-81696-1 [consulta: juny 2023]. Disponible a: https://conwaylife.com/book

10 Annexos

10.1 *Game of Hard Life*

Al fitxer `game-of-hard-life.zip` hi ha el contingut del projecte git on s'ha desenvolupat la primera versió del programa *Game of Hard Life*, producte principal d'aquest treball.

També es troba disponible en línia de forma pública al portal de GitLab:

Game of Hard Life
https://gitlab.com/andreu.bio/game-of-hard-life

Els seus components són:

main.py Aquest pot aixecar una instància configurada a partir de `settings.py`, i que es controla de forma bàsica amb la barra espaiadora per saltar a la següent generació o amb el botó d'`Enter` per activar el mode automàtic fins que es detecta un estat cíclic sobre tota la graella.

Per cada execució, es genera un fitxer amb el respectiu registre històric en format YAML dins el directori `./tests`

settings.py Permet ajuntar el comportament de l'entorn. Actualment, es disposen les següents variables:

pixel, X, Y: Configura la mida de la graella (`World`) i els pixels amb els que es representa cada cel · la.

PACE: Temps mínim que ha de tardar el mode automàtic entre els passos de cada generació

HARD: Activa el mode *Game of Hard Life*, per tant, la caducitat de les cel · les es tenen en compte.

LIFESPAN: Nombre de generacions que es manté la vida útil de les cel · les (caducitat).

INIT_MARGIN: Marge que es dona entre la mostra de cel · les aleatòires i els marges de la graella.

TOROIDAL: Activa el mapa com a projecció sobre superfície toroidal.

HSIZE: Determina l'extensió de l'històric de cel · les dins el model de registre `Registry`.

INF_LOOP: Determina si un cop detectat un bucle cíclic continua executant indefinidament les següents generacions (per defecte: *False*).

RULESTRING: Escull el descriptor de regles de naixement i supervivència de les cel · les.

KERNEL: Escull un vector de distribució de distàncies.

forms.py Fitxer destinat a guardar formes notables que poden ser incorporades des del fitxer de `settings.py` com estat inicial.

rulestrings.py Fitxer destinat a guardar combinacions de regles de naixement i supervivència notables que poden ser incorporades des del fitxer de `settings.py` com estat inicial.

agent.py Esquelet de l'agent que pot realitzar execucions independents.

A la fase final s'espera que sigui capaç d'obtenir decisions a partir de les dades que s'extreuen d'aquestes execucions.

analyze.ipynb *Jupyter Notebook* on es presenten les gràfiques d'anàlisi dels resultats obtinguts a la ruta `tests` produïdes a les fases d'experimentació.

10.2 Dades experimentals

Al fitxer `experiment.zip` s'hi agrupa la col · lecció de mostres i dades experimentals referents a la recopilació corresponent a les tasques:

- A-01, a partir de l'entorn base *Game of Life*
- A-03, a partir de l'entorn *Game of Hard Life - betta*
- A-05, a partir de l'entorn *Game of Hard Life - 0.3*
- A-07, a partir de l'entorn *Game of Hard Life - 1.0*

Addicionalment, s'hi inclou el fitxer `tests.zip` que conté els fitxers de tipus YAML resultat dels experiments fets a les darreres tasques d'anàlisi, i que han estat el recurs de consum per part del Jupyter Notebook `analyze.ipynb`.

10.3 Document: Seguiment del treball

El document `andreubio_informe_seguiment_fase-final.pdf` recull el seguiment complet del treball, que completa la informació referent a la secció 2.4 Planificació del Treball.

En aquest es detalla cada una de les tasques, també es descriuen les activitats no previstes i les relacions de desviació en la temporalització del projecte, així com les accions de mitigació.