UNIVERSITATEA "ALEXANDRU IOAN CUZA" DIN IAȘI
FACULTATEA DE INFORMATICĂ

LUCRARE DE LICENȚĂ

Tehnici pentru crearea rețelelor de sortare

propusă de

Ionuț-Lucian Dominteanu

Sesiunea: *februarie, 2020*

Coordonator științific
Lector Dr. Cristian Frăsinaru

UNIVERSITATEA "ALEXANDRU IOAN CUZA" DIN IAȘI
FACULTATEA DE INFORMATICĂ

Tehnici pentru crearea rețelelor de sortare

Ionuț-Lucian Dominteanu

Sesiunea: *februarie, 2020*

Coordonator științific
Lector Dr. Cristian Frăsinaru

Avizat,

Îndrumător Lucrare de Licenţă

Titlul, Numele şi prenumele _____

Data _____ Semnătura _____

DECLARAȚIE privind originalitatea conținutului lucrării de licență

Subsemntatul(a) Ionuț-Lucian Dominteanu domiciliul în Bacău strada Lalelelor bloc 3 scara A apartament 4, născut(ă) la data de 7.11.1997, identificat prin CNP 1971107046262, absolvent(a) al(a) Universității „Alexandru Ioan Cuza" din Iași, Facultatea de informatică specializarea informatică, promoția. 2016-2019, declar pe propria răspundere, cunoscând consecințele falsului în declarații în sensul art. 326 din Noul Cod Penal și dispozițiile Legii Educației Naționale nr. 1/2011 art.143 al. 4 si 5 referitoare la plagiat, că lucrarea de licență cu titlul „Tehnici pentru crearea rețelelor de sortare", elaborată sub îndrumarea dl. / d-na lector dr. Cristian Frăsinaru, pe care urmează să o susțină în fața comisiei este originală, îmi aparține și îmi asum conținutul său în întregime.

De asemenea, declar că sunt de acord ca lucrarea mea de licență să fie verificată prin orice modalitate legală pentru confirmarea originalității, consimțind inclusiv la introducerea conținutului său într-o bază de date în acest scop.

Am luat la cunoștință despre faptul că este interzisă comercializarea de lucrări științifice in vederea facilitării falsificării de către cumpărător a calității de autor al unei lucrări de licență, de diploma sau de disertație și în acest sens, declar pe proprie răspundere că lucrarea de față nu a fost copiată ci reprezintă rodul cercetării pe care am întreprins-o.

Dată azi, …… Semnătură student ……

DECLARAŢIE DE CONSIMŢĂMÂNT

Prin prezenta declar că sunt de acord ca Lucrarea de licenţă cu titlul „*Tehnici pentru crearea reţelelor de sortare*”, codul sursă al programelor şi celelalte conţinuturi (grafice, multimedia, date de test etc.) care însoţesc această lucrare să fie utilizate în cadrul Facultăţii de Informatică.

De asemenea, sunt de acord ca Facultatea de Informatică de la Universitatea „Alexandru Ioan Cuza” din Iaşi, să utilizeze, modifice, reproducă şi să distribuie în scopuri necomerciale programele-calculator, format executabil şi sursă, realizate de mine în cadrul prezentei lucrări de licenţă.

Iaşi, *6.02.2020*

Absolvent Ionuţ-Lucian Dominteanu

_____

CUPRINS

INTRODUCERE

Aplicația prezentată în această lucrare are ca scop oferirea unui mediu de lucru cu rețele de sortare, principalele funcționalități fiind generarea, vizualizarea și testarea corectitudinii și performanței acestora. Rețele de sortare sunt o categorie de algoritmi foarte importanți în rezolvarea problemei ordonării de valori. O caracteristică distinctă a acestor algoritmi este faptul că datele de intrare nu modifică comparațiile efectuate.

Studiul rețelelor de sortare a fost început în 1954, de către Nelson Raymond, O'Conner Daniel și P.N. Armstrong[1], iar în 1973 Donald Knuth, a publicat al treilea volum al cărții „The Art of Computer Programming", în care a prezentat teoria aplicării lor în hardware, sugerând folosirea lor in construcția de rețele de comutatori. În practică, putem găsi rețelele de sortare utilizate în plăcile video[2] și calculatoarele multiprocesor, în hardware, și în securizarea calculelor multiparty[3], în software. Două proprietăți motivează alegerea lor în unitățile de procesare grafică, abilitatea lor de a accepta executarea în paralel și faptul că timpul de execuție este foarte predictibil, spre deosebire de alți algoritmi de sortare, precum quicksort.[2] Având în vedere cât de mult a crescut industria plăcilor video în ultimii ani, rețele de sortare au căpătat mai multa importanță.

Algoritmul de generare a rețelelor de sortare pe care se concentrează această lucrare este Batcher odd-even mergesort, o metodă de combinare a rețelelor concepută de Kenneth Edward Batcher, fost profesor la Kent State University. Acest algoritm își face și el apariția în cartea lui Donald Knuth, dar de asemenea se regăsește și în a doua carte GPU Gems, fiind unul din algoritmii recomandați pentru sortarea folosind unitățile de procesare grafica.[2]

Primul capitol va prezenta teoria rețelelor de sortare, caracteristici, algoritmi de construcție și metode de testare.

Al doilea capitol va prezenta aplicația, descrierea elementelor de interfața și a funcționalităților puse la dispoziție utilizatorilor.

Al treilea capitol va descrie în detaliu implementarea, tehnologiile folosite, arhitectura aplicației și diferiții algoritmi folosiți în realizarea acesteia.

[1] https://worldwide.espacenet.com/patent/search/family/024573610/publication/US3029413A
[2] https://developer.nvidia.com/gpugems/GPUGems2/gpugems2_chapter46.html
[3] U. Maurer. Secure multi-party computation made simple.

1 REȚELE DE SORTARE

1.1 INTRODUCERE

O rețea de sortare este o rețea de comparatori în care rezultatul produs este mereu o listă monoton crescătoare. O rețea de comparatori este un algoritm care primește la intrare o lista de valori comparabile și returnează la ieșire o permutare a acestei liste.

Rețele de comparatori conțin două tipuri de elemente, fire și comparatori. Fiecare fir în parte primește o valoare din secvența oferită ca date de intrare. Fiecare comparator conectează o pereche de fire, așadar are două valori la intrare, valori ce vor fi interschimbate între fire, în caz că nu sunt ordonate. La fiecare moment dat, un fir poartă o singură valoare, iar datorită interschimbărilor executate de comparatori, valorile purtate de fire la ieșire vor forma o permutare a secvenței de intrare. O rețea de sortare conține o colecție de comparatori ce asigura că permutarea rezultată va fi mereu ordonată.

Grafic, un comparator poate avea următoarea reprezentare:

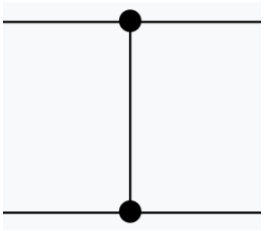


Figura 1 Comparator

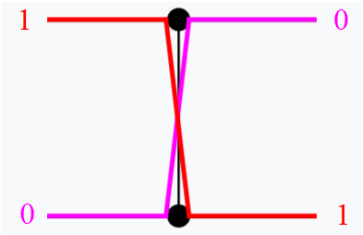


Figura 2 Comparator cu valori

Folosind figurile de mai sus, putem forma o reprezentare grafică a unei rețele de comparatori astfel:

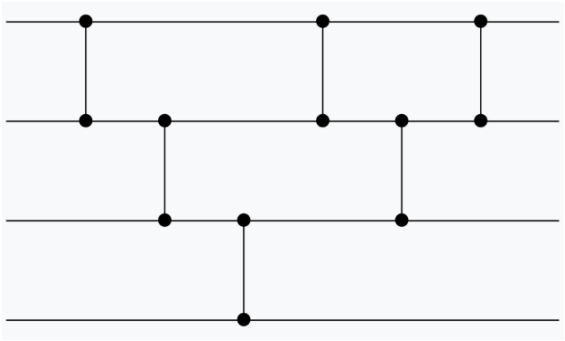


Figura 3 Retea de sortare

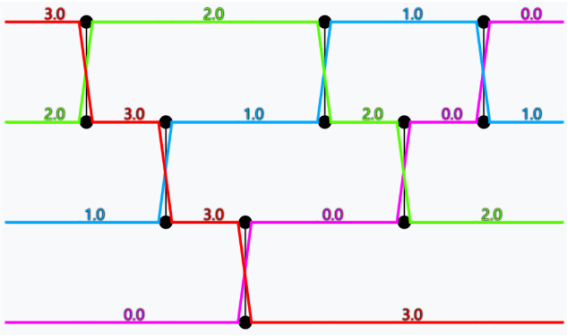


Figura 4 Retea de sortare cu valori

În figura 3, liniile orizontale reprezintă firele, iar figurile formate din linii verticale și discuri reprezintă comparatorii. În figura 4 se regăsește rețeaua din figura 3, dar împreună cu rezultatul

unui calcul. Fiecare linie colorată reprezintă o valoare, roșu pentru 3, verde pentru 2, albastru pentru 1 și mov pentru 0, iar în dreptul fiecărui comparator valorile sunt interschimbate astfel încât valoare mai mică să se poziționeze pe firul superior.

Câteva avantaje ale rețelelor de sortare:

- Pot fi implementate atât în software, cât și în hardware
- Permit executarea de comparații în paralel
- Numărul de comparații este același, indiferent de datele de intrare
- Timpul de execuție variază foarte puțin
- Ocupa puțină memorie în timpul execuției

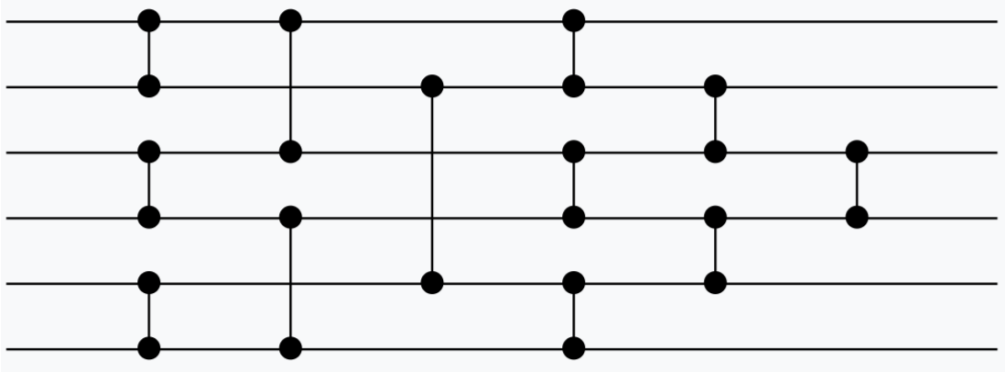
1.2 REȚELE OPTIME

Rețele de sortare optime pot fi cu adâncime minimă, maximizând astfel execuția în paralel, sau dimensiune minimă, cu cat mai puțini comparatori. Aceste rețele sunt foarte utile în algoritmi de construcție recursiva, folosind ca pas de baza returnarea uneia dintre rețelele optime.

Primele rețele de sortare au fost descrise într-un brevet înregistrat de P.N. Armstrong, R.J. Nelson și D.J. O'Connor în 1954[4]. De asemenea, acesta conținea și rețele exemplu pentru n = [4, 8], fiind formate din 5, 9, 12, 18 și respectiv 19 comparatori. Pentru construirea acestor rețele s-au folosit de regula $\hat{S}(n + 1) \leq \hat{S}(n) + n$, unde $\hat{S}(n)$ reprezintă numărul minim de comparatori într-o rețea de dimensiune n.

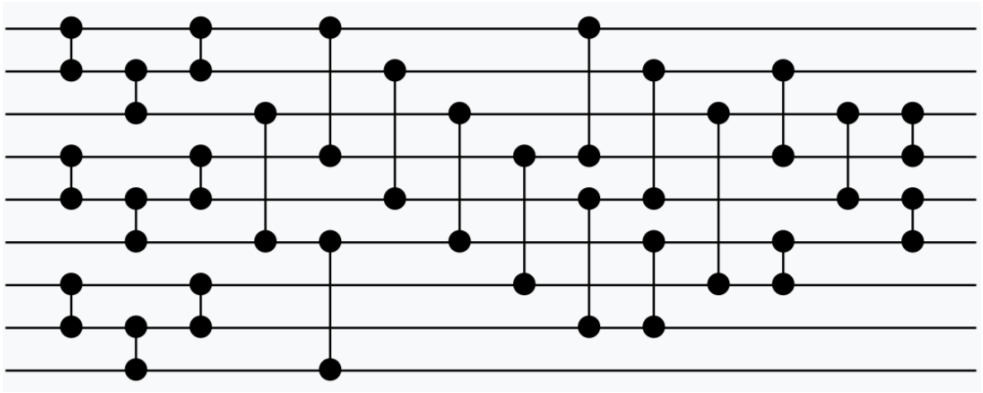
Teoria limitei lui $\hat{S}(n)$ a fost explorată în continuare de R.J. Nelson împreuna cu R.C. Bose[5], atunci când au demonstrat că $\hat{S}(2^n) \leq 3^n - 2^n$, publicând rezultatul în 1962. Doi ani mai tarziu, R.W. Floyd și D. E. Knuth[6] au ajuns la formula $\hat{S}(n) = O(n^{1+\frac{c}{\sqrt{\log n}}})$. În 1968, K.E. Batcher a publicat algoritmul odd-even mergesort și demonstrația faptului că $\hat{S}(n) = O(n(\log n)^2)$, astfel rețele optime pentru 7 și 8 au fost stabile ca având 16, și respectiv 19, comparatori.[7]

[4] https://worldwide.espacenet.com/patent/search/family/024573610/publication/US3029413A
[5] Bose, R. C.; Nelson, R.J. (1962). "A sorting problem". Journal of the ACM Volume 9
[6] Notices of the American Mathematical Society (1967) pg. 283
[7] Batcher, K. E. (1968). "Sorting networks and their applications". Proc. AFIPS Spring Joint Computer Conference. pg. 307-314



Rețeaua de sortare optimă pentru n = 6

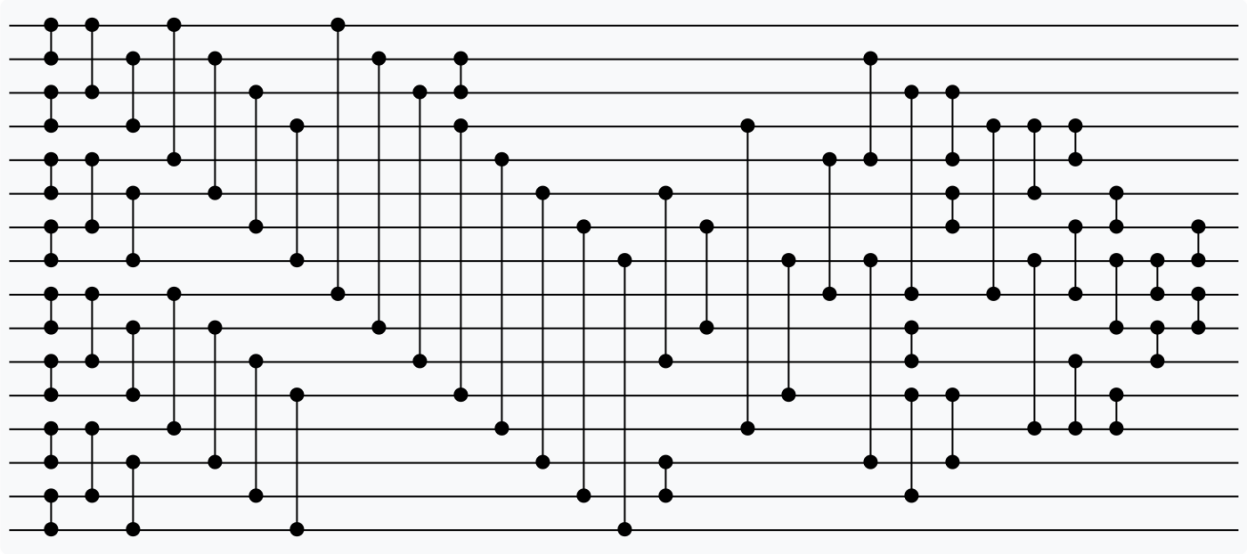
Începând cu n = 9 inclusiv, algoritmul lui Batcher nu mai produce rețele de sortare optime. Rețeaua de dimensiune 9 conținând 25 de comparatori a fost descoperită de R.W. Floyd în 1964 și rețeaua pentru n = 10 cu 29 de comparatori a fost găsită de către Waksman în 1969, optimalitatea celor doua rețele fiind demonstrată în 2014.[8]



Rețeaua de sortare optimă pentru n = 9

O rețea cu 35 de comparatori pentru dimensiunea 11 a fost descoperită în 2019 de către Jannis Harder, demonstrând astfel că limita inferioara pentru n = 12 este 39 de comparații.[9] Pentru dimensiunea 13, o rețea cu 45 de comparatori a fost construită folosind un algoritm genetic, realizat de Hughes Juillé în 1995. Pentru n = 16, există o rețea de sortare cu 60 de comparații, descoperită de M.W. Green, iar împreună cu G. Shapiro a contribuit la rețeaua cu 39 de comparatori pentru dimensiunea 12.

[8] Codish, Michael; Cruz-Filipe, Luís; Frank, Michael; Schneider-Kamp, Peter (2014). Twenty-Five Comparators is Optimal when Sorting Nine Inputs (and Twenty-Nine for Ten)
[9] https://github.com/jix/sortnetopt



Reteaua de n = 16 și 60 de comparatori descoperita de Greeen

Optimalitatea pe adâncime până în n = 8 a fost demonstrată de Robert W. Floyd și Donald Knuth în anii 1960, pentru n = 9 și 10 în anul 1991[10] într-o lucrare scrisă de Ian Parberry, iar restul până la n = 16 de către Bundala D. și Závodný, J. în 2014.[11]

Următorul tabel prezintă dimensiunile rețelelor optime până în n = 17:[12]

n	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**
Adâncime	0	1	3	3	5	5	6	6	7	7	8	8	9	9	9	9	10
Dimensiune, limita superioara	0	1	3	5	9	12	16	19	25	29	35	39	45	51	56	60	71
Dimensiune, limita inferioara													41	45	49	53	57

[10] Parberry, Ian (1991). "A Computer Assisted Optimal Depth Lower Bound for Nine-Input Sorting Networks". Mathematical Systems Theory

[11] Bundala, D.; Závodný, J. (2014). Optimal Sorting Networks. Language and Automata Theory and Applications. Lecture Notes in Computer Science

[12] Knuth, D. E. (1997). The Art of Computer Programming, Volume 3: Sorting and Searching

1.3 TESTARE

Pentru orice rețea de lungime n exista n! posibile configurații ale datelor de intrare, așadar, în cazul rețelelor ce nu permit demonstrarea teoretică a corectitudinii, testarea pentru dimensiuni mari necesită mult timp. Acest număr poate fi redus la 2^n prin folosirea principiului zero-unu.

Principiul zero-unu declară că o rețea capabila sa sorteze toate secvențele de 0 și 1 este validă pentru orice date de intrare.

Demonstrație.

Daca o rețea de sortare transformă secvența $x = (x_1,x_2,...,x_n)$ în $y = (y_1,y_2,...,y_n)$, atunci va transforma $x = (f(x_1),f(x_2),...,f(x_n))$ în $f(y) = (f(y_1),f(y_2),...,f(y_n))$, pentru orice f, funcție monotonă crescătoare. Deoarece un singur comparator cu date de intrare x și y, va produce min(x, y) pentru firul superior și max(x, y) pentru firul inferior, putem trage concluzia că pentru datele de intrare f(x) și f(y), va produce min(f(x), f(y)) și max(f(x), f(y)), dar cum $f(x) \leq f(y)$ pentru orice $x \leq y$, avem:

min(f(x), f(y)) = f(min(x, y))

max(f(x), f(y)) = f(max(x, y))

Presupunem prin reducere la absurd că există o rețea ce sortează toate secvențele de 0 și 1, dar există o lista de numere oarecare ce nu este ordonată corect. Fie secvența menționată (a_1, a_2, ..., a_n) cu $a_i < a_j$, dar datele de ieșire plasează a_i după a_j.

Fie f funcția monotonă crescătoare definită astfel: $f(x) = \begin{cases} 1, daca\ x > y_i \\ 0, daca\ x \leq y_i \end{cases}$

Dacă pentru secvența de intrare (a_1, a_2, ..., a_n) avem a_j înainte de a_i la ieșire, pentru ($f(a_1)$, $f(a_2)$, ..., $f(a_n)$) la intrare, avem $f(a_j)$ înainte de $f(a_i)$ la ieșire, dar cum $f(a_j) = 1$ și f(ai) = 0 contrazicem afirmația inițială cum că rețeaua sortează orice secvența de 0 și 1.

1.4 CREARE

Printre cei mai eficienți algoritmi de construcție a rețelelor de sortare folosiți în practică găsim Batcher odd-even mergesort, bitonic sort, Shell sort și pairwise sorting network, producând rețele de adâncime $O(\log^2 n)$ și dimensiune $O(n \log^2 n)$. Pentru adâncimi mai mici se pot folosi rețele

AKS, descoperite de Ajtai, Komlós, and Szemerédi[13], dar acestea oferă putină practicabilitate, datorită constantei de dimensiuni mari[14]. Există și un algoritm pentru construcția rețelelor de dimensiune O(n log n), descoperit de Michael T. Paterson, dar adâncimea O(n log n) limitează potențialul lor practic.

1.4.1 Bubble sort și insertion sort

Două metode ușor de implementat pentru construcția de rețele de sortare pot fi obținute prin adaptarea algoritmilor bubble sort și insertion sort, urmărind principiile de selecție, respectiv de inserție.

Putem privi algoritmul insertion sort ca o secvență de sortări a unor liste progresiv mai mari, începem cu două valori, apoi trei, și repetăm până la n.

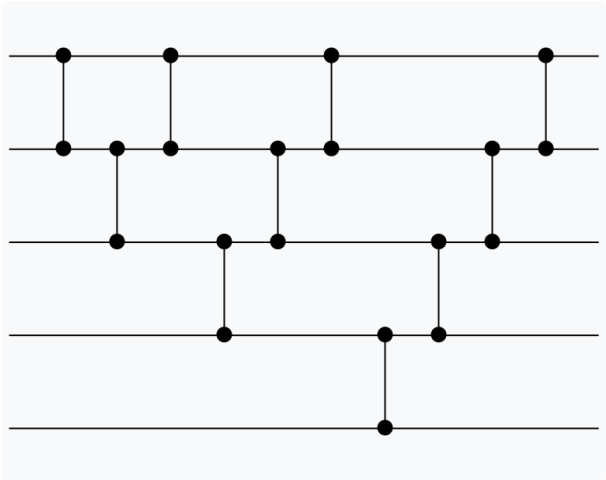


Figura 5 Insertion sort

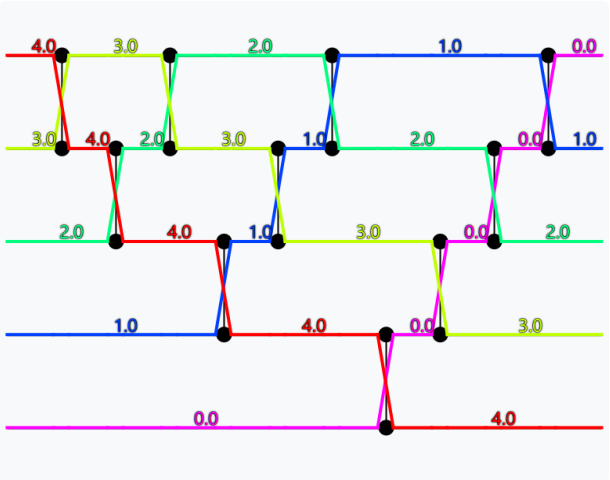


Figura 6 Insertion sort cu valori

Algoritmul bubble sort începe cu sortarea ultimului element, apoi penultimul element, adăugând elemente corect sortate la sfârșitul listei până când toata secvența este ordonată.

[13] Ajtai, M.; Komlós, J.; Szemerédi, E. (1983). An O(n log n) sorting network
[14] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L. (1990). Introduction to Algorithms

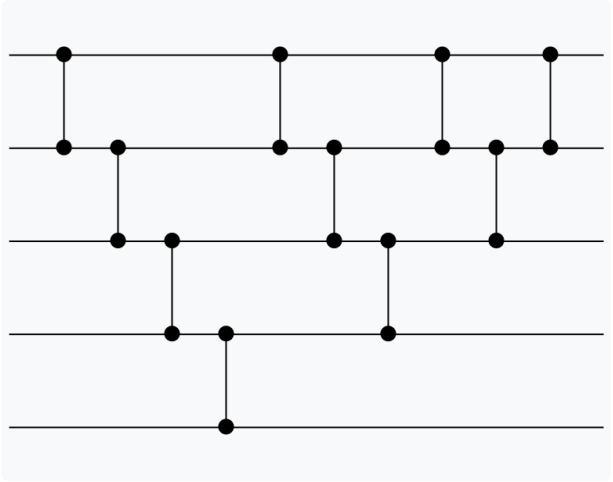


Figura 7 Bubble sort

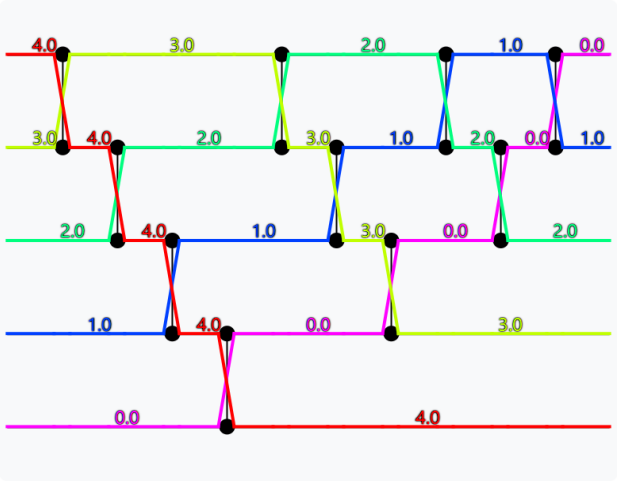


Figura 8 Bubble sort cu valori

Atunci când nu permitem comparatori în paralel, bubble sort produce oglinda rețelei construite de către insertion sort. În schimb, atunci când permitem paralelism, cei doi algoritmi produc aceeași rețea.

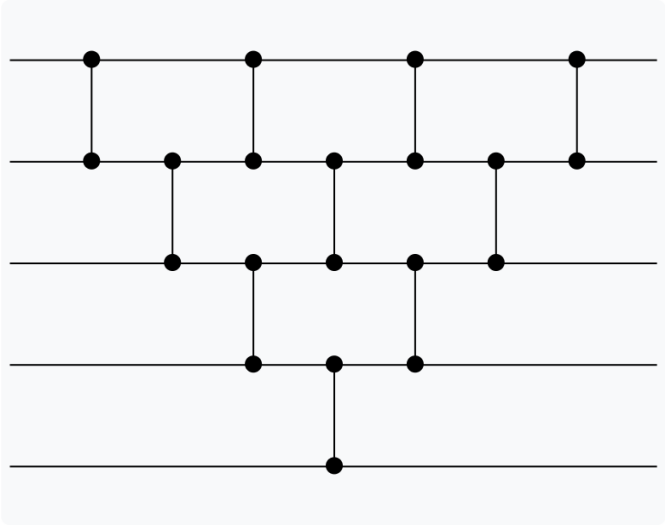


Figura 9 Bubble/Insertion sort cu paralelism

1.4.2 Batcher odd-even mergesort

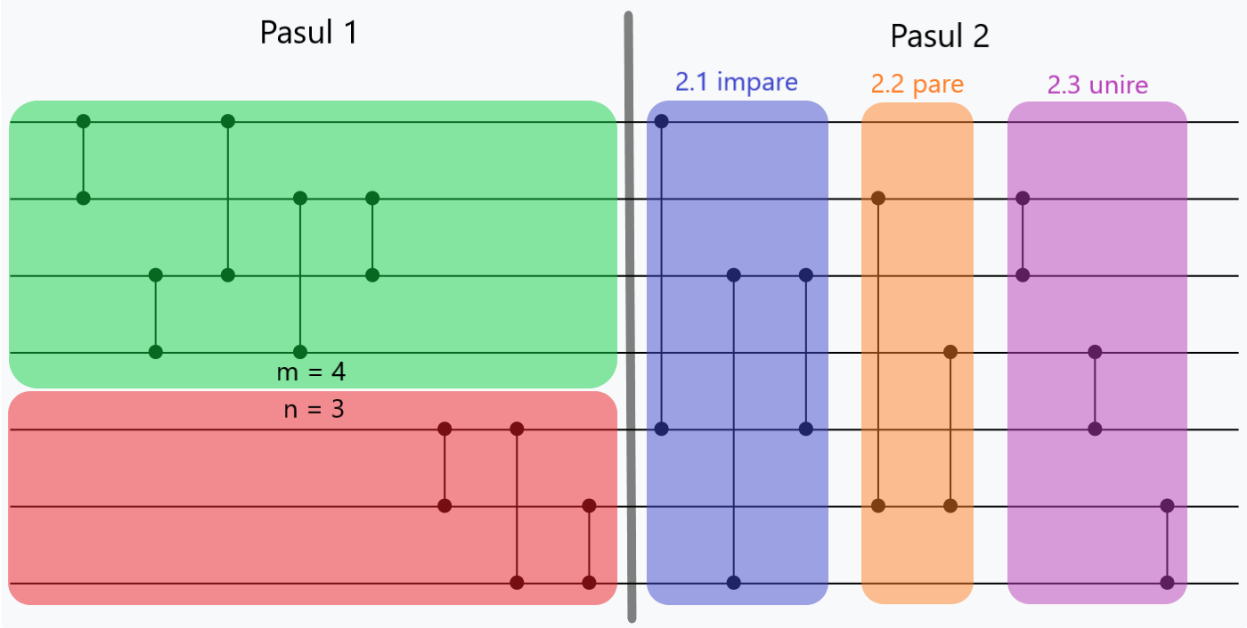
Metoda lui Batcher poate fi folosită atât pentru combinarea a doua rețele existente, cât și pentru a genera o rețea completă, de lungime O(n (log n)2) și adâncime O((log n)2), n fiind dimensiunea listei sortate.

Generarea unei rețele noi are 2 pași:

1. Împărțim rețeaua în două și apelăm algoritmul recursiv pentru fiecare rețea rezultată.
2. Combinam cele doua rețele rezultate.

Combinarea a două rețele are 3 pași:

1. Aplicăm sortarea peste lista formată din elementele pe poziție impară din cele două rețele.
2. Similar cu pasul 1, dar pentru elemente de pe poziție pară.
3. Adăugam câte un comparator (i, i + 1) pentru fiecare (i = 0, i < n – 1, i += 2)



În figura de mai sus se poate observa o rețea de lungime 7, generată de algoritmul Batcher odd-even mergesort. Colorate cu verde și roșu regăsim cele doua subrețele generate prin recursivitate, acestea constituind primul pas al algoritmului. În partea a doua, colorate cu albastru și portocaliu avem rețeaua valorilor de pe poziții impare, respectiv pare. Ultima subrețea, colorată cu mov, unește cele doua subrețele anterioare.

Demonstrația algoritmului de combinare.

Notăm rezultatul primei rețele cu (x_1, x_2, ..., x_m) și rezultatul următoarei rețele cu (y_1, y_2,..., y_n). Secvența de valori impare formată din sortarea listelor (x_1, x_3,...,$x_{2\lceil m/2\rceil-1}$) și (y_1, y_3,...,$y_{2\lceil n/2\rceil-1}$) se va nota cu v = (v_1, v_2,..., $v_{\lceil m/2\rceil + \lceil n/2\rceil}$). Similar se va nota cu w = (w1, w2,...,$w_{\lfloor m/2\rfloor + \lfloor n/2\rfloor}$) sortarea secvențelor de valori impare (x_2, x_4,...,$x2_{\lfloor m/2\rfloor}$) și (y_2, y_4,...,$y_{2\lfloor n/2\rfloor}$).

După construirea secvențelor v și w aplicam următoarele operații de interschimbare:

$$w_1{:}v_2,\ w_2{:}w_3,\ w_3{:}v_4,\ \dots\ ,w_{\lfloor m/2\rfloor + \lfloor n/2\rfloor}{:}v^{*} \quad (1)$$

peste secvența:

$$(v_1, w_1, v_2, w_2, \dots\ , v_{\lfloor m/2\rfloor + \lfloor n/2\rfloor}, w_{\lfloor m/2\rfloor + \lfloor n/2\rfloor}, v^{*}, v^{**}) \quad (2)$$

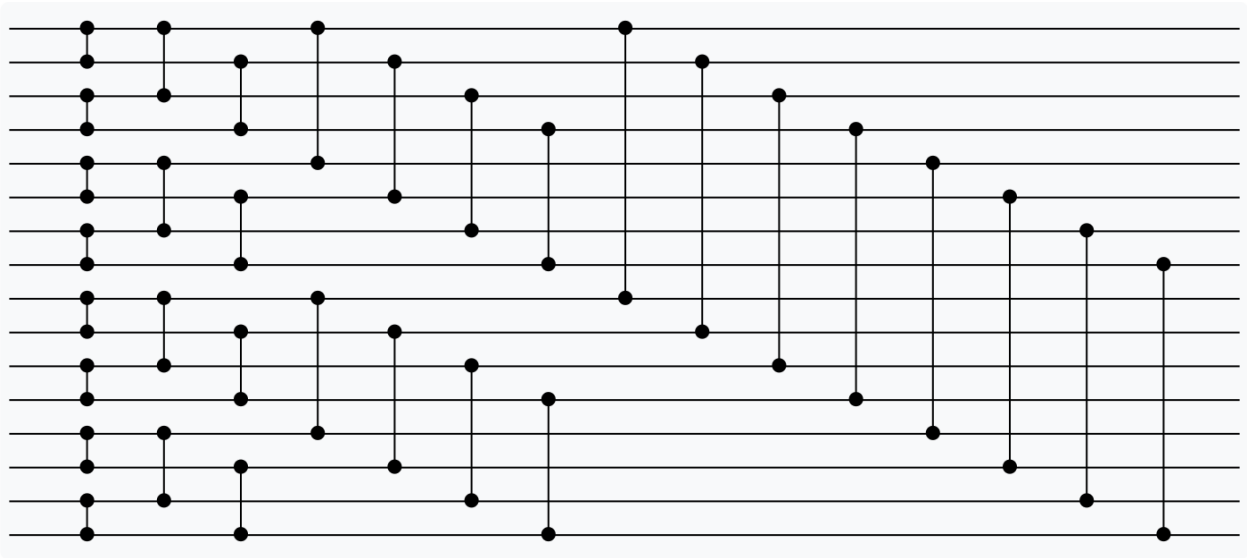
Atunci când și m și n sunt pare, $v^{*} = v_{\lfloor m / 2 \rfloor + \lfloor n / 2 \rfloor + 1}$ nu există, iar $v^{**} = v_{\lfloor m / 2 \rfloor + \lfloor n / 2 \rfloor + 1}$ există doar când m și n sunt amândouă impare. Dacă aplicăm principiul zero-unu, $(x_1,..., x_m)$ va conține k de 0 și m – k de 1, iar $(y_1,...,y_n)$ va conține h de 0 și n – h de 1, pentru niște k și h. Rezultă că $(v_1, v_2, ...)$ va conține $\lceil k / 2 \rceil + \lceil h / 2 \rceil$ de 0, urmat de restul de 1, iar $(w_1, w_2, ...)$ va fi format din $\lfloor k / 2 \rfloor + \lfloor h / 2 \rfloor$ de 0 și restul de 1.

$$(\lceil k / 2 \rceil + \lceil h / 2 \rceil) – (\lfloor k / 2 \rfloor + \lfloor h / 2 \rfloor) = 0, 1 \text{ sau } 2$$

Pentru diferența de 0 și 1 secvența (2) este deja în ordine, iar pentru 2, interschimbările din (1) vor sorta. Astfel secvența rezultată este sortată.

1.4.3 Filtre pentru algoritmii genetici

Primii algoritmi genetici generatori de rețele de sortare au fost folosiți de către W.D. Hillis în 1992. Prima rețea construita a fost pentru n = 16 și conținea 61 de comparatori. În următoarele rețele Hillis a descoperit că majoritatea încep cu aceeași secvența de 32 de comparatori, comparatori regăsiți și in prima parte a rețelei lui Green. Această secvență, denumită filtrul lui Green, produce doar 151 de secvențe binare nesortate, fiind astfel foarte folositoare ca punct de plecare pentru algoritmii genetici.[15]



Filtrul lui Green

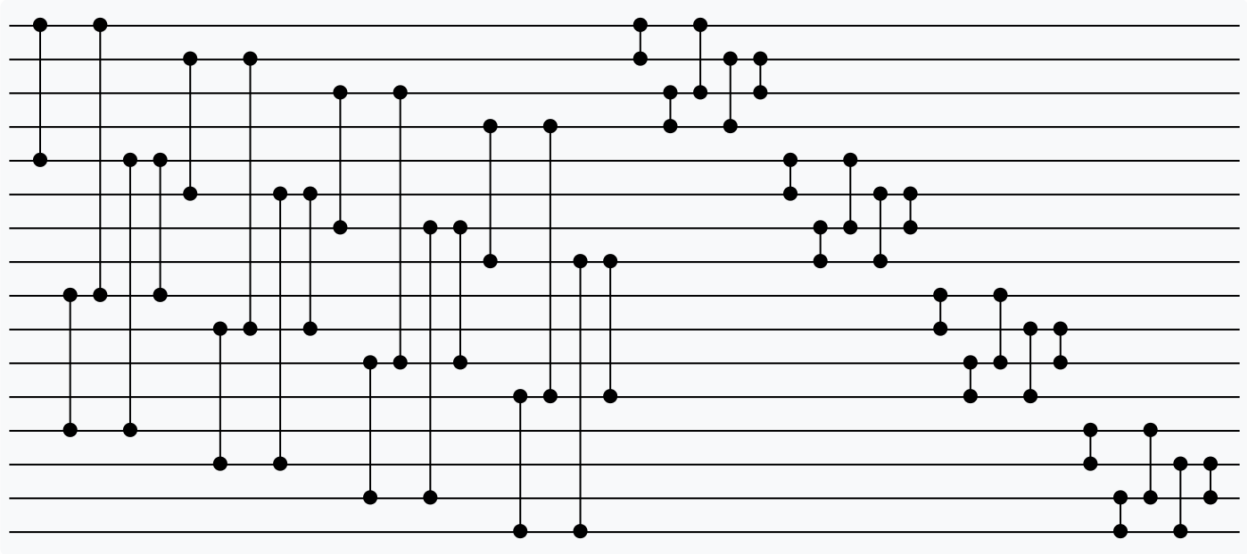
Un alt filtru mai eficient, ce produce doar 53 de secvențe binare nesortate, poate fi construit folosind o matrice de n x n cu fiecare element $M_n(i, j) = i * n + j$ pentru $0 \leq i, j \leq n - 1$, unde n^2 este

[15] Drue Coles (2012). Efficient filters for the simulated evolution of small sorting networks

numărul de fire al rețelei de sortare. Pentru fiecare coloană din M_n adăugam comparatorii de pe firele indexate de valorile coloanei, iar apoi aplicăm similar pentru rândurile din M_n. Acest filtru a fost prezentat de Drue Coles in 2012 și a fost denumit Square Filter.[16]

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{pmatrix}$$

M_4 folosit pentru generarea unui Square Filter



Square filter pentru 16 fire

1.5 CONCLUZIILE CAPITOLULUI

In acest capitol a fost prezentată teoria rețelelor de sortare, începând cu structura, caracteristici și reprezentarea lor grafica, continuând cu istoricul rețelelor optime și exemple relevante, prezentarea principiului zero-unu și demonstrația acestuia și încheind cu metode de generare, trei algoritmi ce produc rețele întregi și două filtre folosite in algoritmii genetici.

[16] Drue Coles (2012). Efficient filters for the simulated evolution of small sorting networks

2 ARHITECTURA ȘI IMPLEMENTARE

2.1 TEHNOLOGII UTILIZATE

2.1.1 Java

Java este un limbaj de programare orientat-obiect, puternic tipizat, care are ca scop principal "write once, run anywhere". A fost conceput de James Gosling când lucra la Sun Microsystems la începutul anilor '90 și apoi lansat în 23 mai 1995. Până în 2018, Github plasa Java ca al doilea cel mai popular limbaj de programare după numărul de repositories[17] și tot locul doi după căutările pe Google.[18]

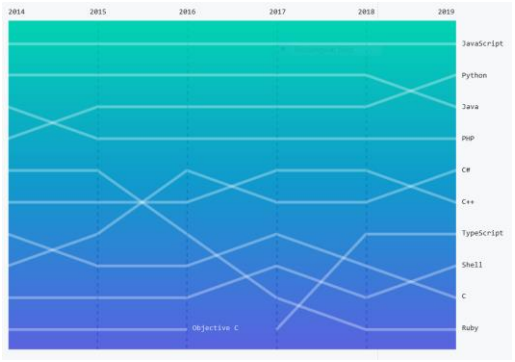


Figure 1 Clasamentul dupa repositories

Worldwide, Jan 2020 compared to a year ago:

Rank	Change	Language	Share	Trend
1		Python	29.72 %	+4.3 %
2		Java	19.03 %	-1.9 %
3		Javascript	8.2 %	+0.1 %
4		C#	7.28 %	-0.2 %
5		PHP	6.09 %	-1.1 %
6		C/C++	5.91 %	-0.3 %
7		R	3.72 %	-0.2 %
8		Objective-C	2.47 %	-0.6 %
9		Swift	2.36 %	-0.2 %
10		Matlab	1.79 %	-0.2 %

Figure 2 Clasamentul dupa Google trends

Cele cinci principii al limbajului Java sunt:[19]

1. Trebuie sa fie simplu, orientat-obiect și familiar.
2. Trebuie sa fie robust și sigur.
3. Trebuie sa fie neutru arhitectural și portabil
4. Trebuie sa execute cu performanță înaltă
5. Trebuie sa fie interpretat

Câteva caracteristici ale limbajului Java sunt:

1. Tot codul este scris în interiorul claselor
2. Cu excepția primitivelor, toate tipurile de date sunt obiecte

[17] https://octoverse.github.com/#top-languages
[18] http://pypl.github.io/PYPL.html
[19] "1.2 Design Goals of the Java™ Programming Language". Oracle

3. Java nu suporta supraîncărcarea operatorilor și moștenire multiplă
4. Java folosește un automatic garbage collector
5. Sintaxa este în mare parte influențată de limbajul C++.

Principiul portabilității este realizat prin compilarea limbajului Java în reprezentarea intermediară Java bytecode, care va fi executată de o mașină virtuală. Majoritatea dispozitivelor folosesc Java Runtime Enviroment (JRE), o distribuție gratuită ce include mașina virtuală HotSpot și Java standard library.

2.1.2 JavaFX

JavaFX este o platformă pentru crearea aplicațiilor desktop, care are ca scop înlocuirea lui Swing ca bibliotecă GUI standard al limbajului Java[20]. Odată cu lansarea JDK 11 în 2018, Oracle a inclus JavaFX în OpenJDK.

Sun microsystems a anunțat JavaFX în mai 2007, iar anul următor au publicat planurile de a livra JavaFx pentru browser și desktop pană în al treilea sfert al anului 2008. În luna iulie, dezvoltatorii au putut sa descarce un preview al JavaFX SDK, prima versiune oficiala fiind lansata pe 4 decembrie 2008.

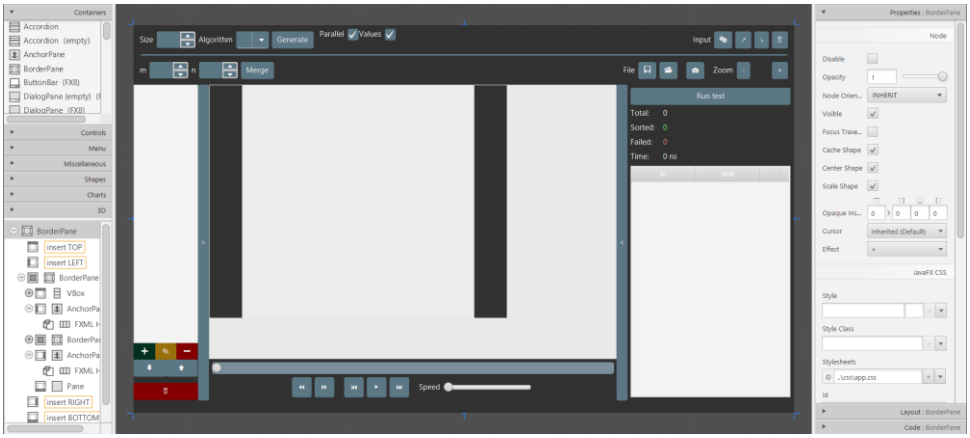
În 10 octombrie 2011 a fost lansat JavaFX 2.0, cu următoarele îmbunătățiri:

- JavaFX Script a fost abandonat și inlocuit cu un nou Java API
- Suport pentru binding de înaltă performanță
- JavaFX runtime specific platformelor
- A fost introdus FXML[21]

2.1.3 SceneBuilder și FXML

JavaFX Scene Builder este o unealtă pentru dezvoltarea aplicațiilor JavaFX în mod vizual. A fost lansată odată cu versiunea 2.1 de JavaFX de către Oracle. Ca multe alte unelte similare, Scene Builder pune la dispoziție funcționalități de drag-and-drop de componente, meniu pentru editarea de proprietăți, o colecție mare de componente predefinite și o fereastră de preview, pentru a vizualiza aplicația fără a fi nevoit sa o rulezi. De asemenea, poate fi integrat cu multe din IDE-urile Java.

[20] https://www.oracle.com/technetwork/java/javafx/overview/faq-1446554.html#6
[21] http://fxexperience.com/wp-content/uploads/2011/08/Introducing-FXML.pdf



Scene Builder se folosește de fișiere FXML pentru a salva datele, un format special de XML creat de Oracle pentru definirea interfeței unei aplicații JavaFX[22]. Extensible Markup Language (XML) este un limbaj de marcare dezvoltat de Consortiul Web.[23] Limbajele bazate pe XML au ca principii extensibilitatea, simplitatea, accesibilitatea și abilitatea de a fi citite și de om și de calculator.[24]

Similar cu HTML și dezvoltarea de aplicații web, FXML permite utilizarea de cascading style sheets pentru modificarea aspectului elementelor de interfața.

2.1.4 Junit 5

Junit este un framework pentru unit testing pentru limbajul de programare Java. Într-un studiu din 2013, peste 10000 de proiecte din Github, Junit s-a plasat pe primul loc ca cea mai folosită bibliotecă externă, fiind inclusă in 30.7% din proiecte.[25]

Odată cu lansarea versiunii Junit 5, proiectul a fost împărțit in 3 module:

1. Junit Platform, conține funcționalitățile de baza ale framework-ului
2. Junit Jupiter, conține noul model de programare și noile funcționalități ale Junit 5
3. Junit Vintaje, facilitează rularea de teste scrise in Junit 3 și Junit 4

[22] http://fxexperience.com/wp-content/uploads/2011/08/Introducing-FXML.pdf
[23] https://www.w3.org/TR/REC-xml/
[24] https://www.w3.org/TR/REC-xml/#sec-origin-goals
[25] http://www.takipiblog.com/2013/11/20/we-analyzed-30000-github-projects-here-are-the-top-100-libraries-in-java-js-and-ruby/

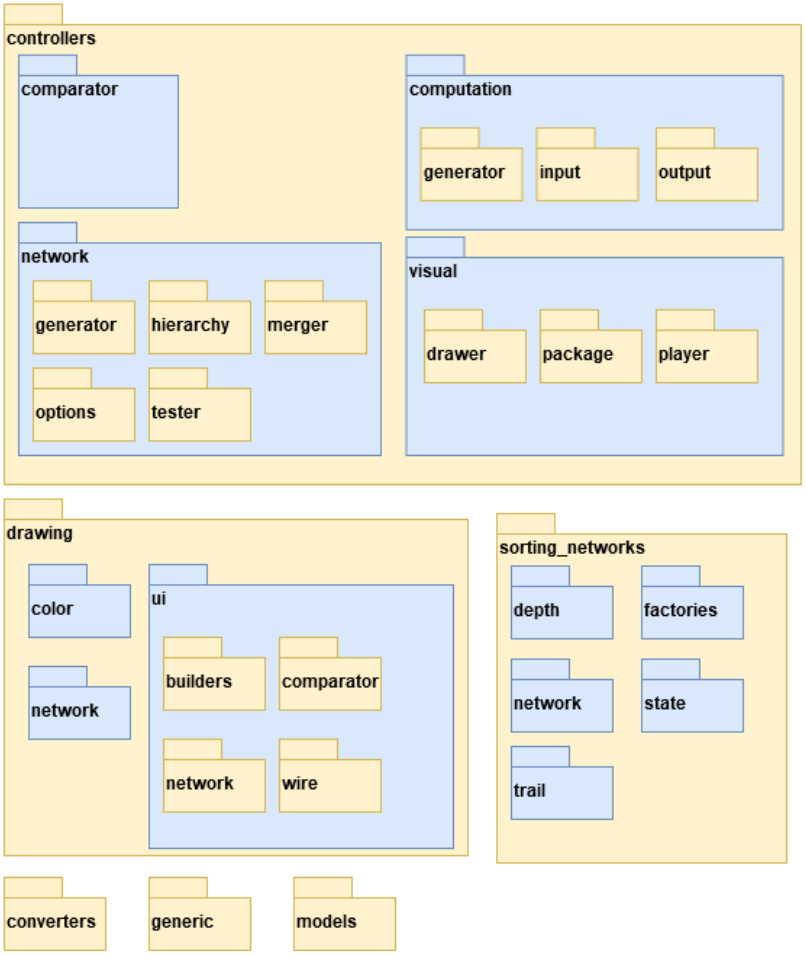
2.2 ARHITECTURĂ

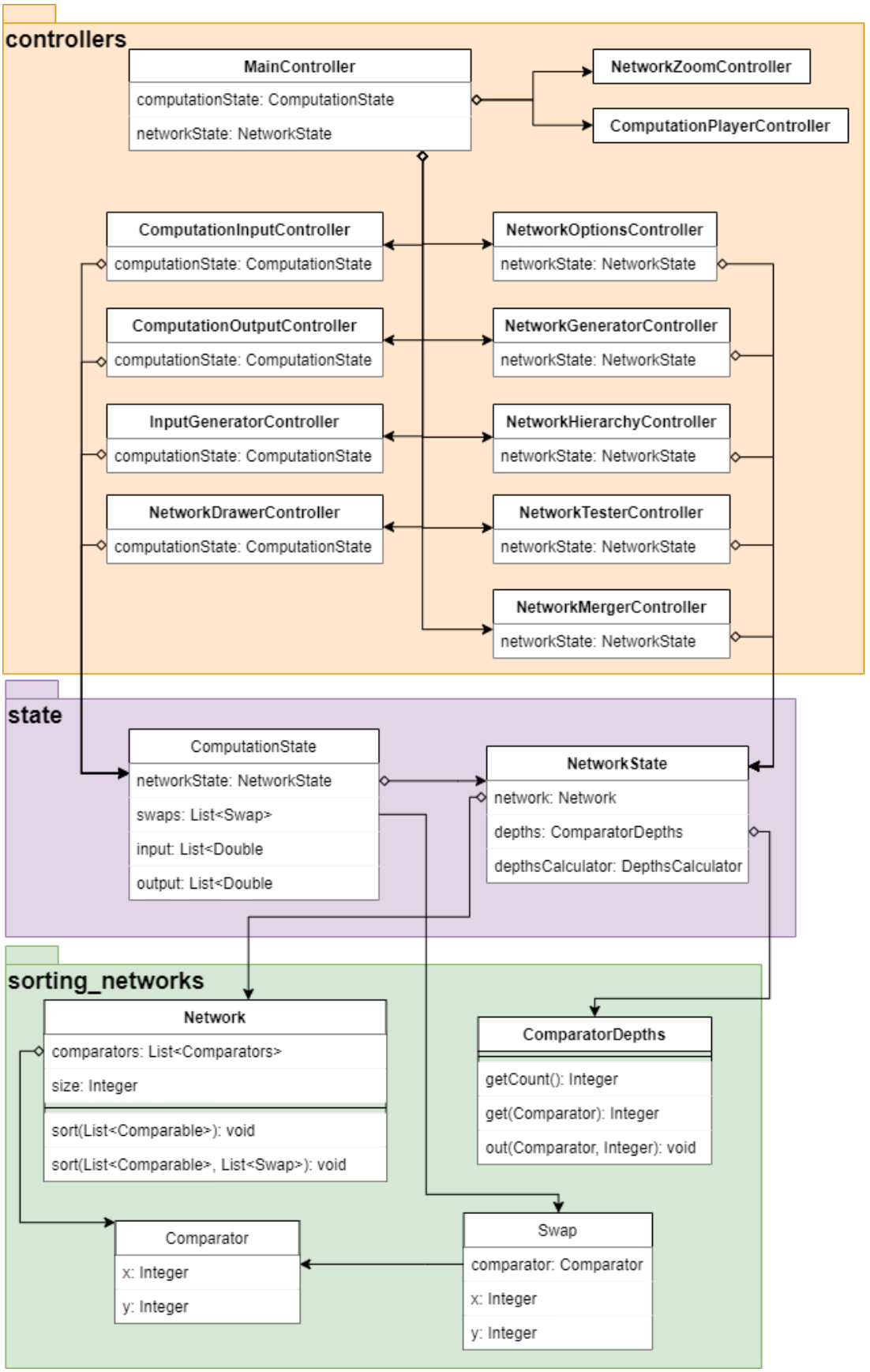
JavaFX Scene Builder folosește pattern-ul MVC, file FXML pentru view și clase Java pentru control. Codul Java al acestei aplicații este împărțit în trei mari pachete:

- controllers – aici se găsesc clasele de control folosite de diferitele componente FXML
- drawing – acest pachet conține toate clasele responsabile cu reprezentarea vizuală a rețelelor de sortare
- sorting_networks – aici se găsesc toate clasele pentru lucrul cu rețele de sortare.

Și alte 3 pachete mai mici cu restul de clase ajutătoare:

- converters
- models
- generic




controllers
MainController
computationState: ComputationState
networkState: NetworkState
NetworkZoomController
ComputationPlayerController
ComputationInputController
computationState: ComputationState
NetworkOptionsController
networkState: NetworkState
ComputationOutputController
computationState: ComputationState
NetworkGeneratorController
networkState: NetworkState
InputGeneratorController
computationState: ComputationState
NetworkHierarchyController
networkState: NetworkState
NetworkDrawerController
computationState: ComputationState
NetworkTesterController
networkState: NetworkState
NetworkMergerController
networkState: NetworkState
state
ComputationState
networkState: NetworkState
swaps: List<Swap>
input: List<Double
output: List<Double
NetworkState
network: Network
depths: ComparatorDepths
depthsCalculator: DepthsCalculator
sorting_networks
Network
comparators: List<Comparators>
size: Integer
sort(List<Comparable>): void
sort(List<Comparable>, List<Swap>): void
ComparatorDepths
getCount(): Integer
get(Comparator): Integer
out(Comparator, Integer): void
Comparator
x: Integer
y: Integer
Swap
comparator: Comparator
x: Integer
y: Integer

Aplicațiile JavaFX pornesc din metoda start(Stage), unde se va apela FXMLLoader și se va încarcă view-ul „main.fxml”, pornind astfel construcția fiecărui controller al componentelor FXML incluse în acesta. Fiecare controller va fi apoi injectat de către FXMLLoader în Main Controller, clasa ce controlează „main.fxml”. În constructorul acestuia se va instanția un obiect NetworkState și unul ComputationState, obiecte ce vor fi apoi comunicate claselor de control corespunzătoare, servind rolul de stare comună pentru toate componentele aplicației. După legarea proprietăților dintre controller-i ce trebuie sincronizați, se va apela generarea unei rețele exemplu.

De fiecare data când o proprietate din clasele de stare este schimbată, toți ascultătorii sunt notificați, iar fiecare controller actualizează interfața componentei pentru care este responsabil.

2.3 DETALII DE IMPLEMENTARE

2.3.1 Clase

Pachetul sorting_networks

Interfața Network este folosită pentru stocare datelor unei rețele de sortare. Conține o lista de comparatori, o dimensiune pentru datele de intrare și doua metode pentru sortare:

- O funcție cu un parametru, lista ce va fi sortata
- O funcție cu doi parametri, lista ce va fi sortata și un parametru de ieșire de tip colecție de obiecte Swap

Clasa Comparator conține doua câmpuri, de tip număr natural, x și y pentru indicele firelor din rețeaua de sortare.

Clasa Swap conține comparatorul unde s-a produs interschimbarea și cele doua valori de pe liniile de date.

Clasa ComparatorDepths conține perechi formate dintr-un comparator și un număr reprezentând adâncimea sa în rețeaua de sortare.

Clasele NetworkState și ComputationState se folosesc pentru stare. Fiecare controller care are nevoie de informații legate de rețeaua de sortare va deține o referință către un obiect de tip NetworkState. În cazul în care este nevoie și de date despre calculul curent se va folosi clasa ComputationState.

NetworkState conține metode de get și set pentru următoarele câmpuri:

- Un obiect Network, pentru rețeaua de sortare curentă

- Un obiect ComparatorDepths, cu adâncimile pentru comparatorii rețelei curente
- Un obiect DepthsCalculator, folosit pentru generarea unui nou ComparatorDepths atunci când Network este modificat

ComputationState conține o referința la un NetworkState și adaugă în plus următoarele câmpuri:

- O listă de obiecte Swap, interschimbările generate în urma calculului curent
- O lista cu datele de intrare
- O lista cu datele de iesire, se va modifica automat cu schimbarea datelor de intrare

Toate aceste câmpuri menționate mai sus folosesc clase din pachetul javafx.beans.property, expunând astfel metode pentru eventualii ascultători.

Pachetul controllers

MainController este controller-ul ferestrei. Acest obiect conține cate o referință pentru fiecare controller al componentelor FXML incluse. La inițializarea acestuia se va instanția un NetworkState și un ComputationState și vor fi transmise claselor controller corespunzătoare. Tot aici se vor lega diverse proprietăți aparținând celorlalte clase controller.

NetworkGeneratorController este controller-ul ce se ocupă cu construirea de noi rețele de sortare. Se folosește de o colecție de implementări ale interfeței NetworkFactory pentru diferitele opțiuni de algoritm.

NetworkMergerController se ocupă de adăugarea de comparatori pentru combinarea subrețelelor de lungime m și n. Acest controller apelează algoritmul Batcher odd-even mergesort.

NetworkDrawerController este responsabil de componenta grafică. Acest controller înregistrează ascultători la proprietățile stării și modifică reprezentarea vizuală atunci când se produc schimbări. Conține o referință la un obiect de tip SortingNetworkDrawer, o interfață cu diferite metode ajutătoare pentru desenare. Expune și o proprietate de timp numar real, utilizată in animarea calculului, câmp ce va fi legat bidirecțional cu câmpul echivalent din ComputationPlayerController.

NetworkHierarchyController conține funcționalități de editare a comparatorilor rețelei de sortare.

Pachetul drawing

SortingNetworkDrawer este o interfață ce expune toate metodele necesare construirii reprezentării vizuale. Conține opțiuni pentru dimensiunea elementelor, culoarea acestora și metode pentru desenarea unei rețele complete sau doar de elemente individuale.

ShapesNetworkDrawer este o implementare a SortingNetworkDrawer ce folosește clase din javafx.scene.shape pentru desenare. O instanța a acestei clase este inițializata în implementarea lui NetworkDrawerController.

WiresColorsPicker este responsabil cu asignarea de culori pentru fiecare linie de valoare din rețeaua de sortare.

NetworkUI este o interfața unde se pot înregistra ascultători pentru elementele vizuale (cum ar fi clicurile peste comparatori).

2.3.2 Algoritmul Batcher odd-even mergesort

Implementarea acestui algoritm se găsește în clasa OddEvenSortFactory, iar referințe la aceasta clase se găsesc în NetworkGeneratorController și NetworkMergerController. Clasa expune două metode pentru generarea de noi rețele:

- Network make(int size) – produce o noua rețea de dimensiune „size” și o returnează
- Network make(Network network, int m, int n) – copiază rețeaua din parametru și adaugă comparatorii necesari combinării subrețelelor de dimensiune m și n (dimensiunea rețelei „network” trebuie se fie egala cu m + n), returnând apoi copia

```
List<Comparator> make(int n) {
   Network network = new Network(n);
   network.comparatori += sortHalves();
   return network();
}
List<Comparator make(Network network, int m, int n) {
   Network newNetwork = network.clone();
   newNetwork.comparatori += merge(m, n);
   return newNetwork();
}
```

```
List<Comparator> sort(int m, int n) {
   comparatori = new List<Comparator>();
   if (m + n <= 2) {
       if (m + n == 2)
            comparatori += new Comparator(0, 1);
       return comparatori;
   }
   if (m == 0)      return sortHalves(n);
   else if (n == 0) return sortHalves(m);
   else {
     comparatori += sortHalves(m);
     comparatori += offset(sortHalves(n), m);
     comparatori += merge(m, n);
   }
   return comparatori;
}
List <Comparator> sortHalves(int n) {
   return sort(⌈n / 2⌉ , ⌊n / 2⌋);
}
List<Comparator> merge(int m, int n) {
   comparatori = new List<Comparator>();
   if (m + n <= 2) {
      if (m == 1 && n == 1)
         comparatori += new Comparator(0, 1);
      return comparatori;
   }
   comparatori += even(merge(⌈m / 2⌉, ⌈n / 2⌉), m, n);
   comparatori += odd(merge (⌊m / 2⌋, ⌊n / 2⌋), m, n);
   comparatori += exchange(m + n);
   return comparatori;
}
List<Comparator> exchange(int n) {
   comparatori = new List<Comparator>();
   for (int i = 1; i < n − 1; i += 2)
      comparatori += new Comparator(i, i + 1);
   return comparatori;
}
```

Functia offset(List<Comparator>, value) va aplica (x + value, y + value) pentru fiecare comparator

Functia even/odd(List<Comparator>, m, n) va converti în valorile corespunzătoare colecției de poziții pare, respectiv impare.

	m = 3			n = 4			
Indice în rețea	0	1	2	3	4	5	6
Indice în subrețea	0	1	2	0	1	2	3
Pare	0		1	2		3	
Impare		0			1		2

Tabelul de mai sus este un exemplu al rezultatelor funcțiilor odd și even pentru m = 3 și n = 4. Funcția odd va primi valori de pe rândul „impare” și le va converti în valorile de pe rândul „Indice în rețea”. Similar funcția even va primi valori de pe rândul „pare”.

2.3.3 Calculul adâncimilor

Pentru atribuirea unei adâncimi fiecărui comparator, astfel încât comparatorii fără fire afectate în comun sa fie plasați în paralel, iar rezultatul sa coincidă cu cel de la rețeaua executată în serie, se va folosi următorul algoritm:

```
ComparatorDepths getDepths(Network network) {
   comparatori = new List<List<Comparator>>();
   comparatori.add(new List<Comparator>());
   for (comparator in network) {
      int start = comparatori.size() - 1;
      for (int depth = start; depth >= 0; depth--) {
         if (overlap(comparatori.get(depth), comparator)) {
            if (comparatori.size() == depth + 1)
               comparatori.add(new List<Comparator>());
            comparatori.get(depth + 1).add(comparator);
            break;
         }
         if (depth == 0) comparatori.get(0).add(comparator);
      }
   }
   return new ComparatorDepths(comparatori);
}
```

Începem prin inițializarea unei liste de liste de comparatori, în care indicele listei părinte reprezintă adâncimea. Pentru fiecare comparator x din rețea, iteram descrescător prin lista părinte și pentru fiecare comparator y de pe aceasta adâncime, verificăm dacă se suprapune cu comparatorul x. Daca găsim suprapuneri la stratul i, plasam comparatorul x pe adâncimea i + 1. Daca nu găsim conflicte pe niciun strat, asignam adâncimea 0 comparatorului x. La sfârșit avem o listă, ai cărei poziții reprezintă adâncimea și ai cărei elemente sunt liste de comparatori.

În implementarea aplicației, funcția „overlap” verifică dacă comparatorii formează segmente ce se intersectează, aceasta metoda produce astfel un model mai clar vizual. Daca singura condiție necesară este găsirea de adâncimi ce nu modifica rezultatul sortării este suficient sa verificăm doar suprapunerea capetelor segmentelor.

2.3.4 Grafică

Se folosesc două containere ce vor servi rolul de straturi, unul pentru desenarea rețelei de sortare și unul pentru desenarea rezultatului sortării. Referințe pentru aceste noduri vor fi injectate în NetworkDrawerController și apoi transmise la SortingNetworkDrawer.

În controller se vor înregistra diferiții ascultători pentru stare, iar atunci când se produc modificări se va șterge desenul și se va apela una din cele două metode:

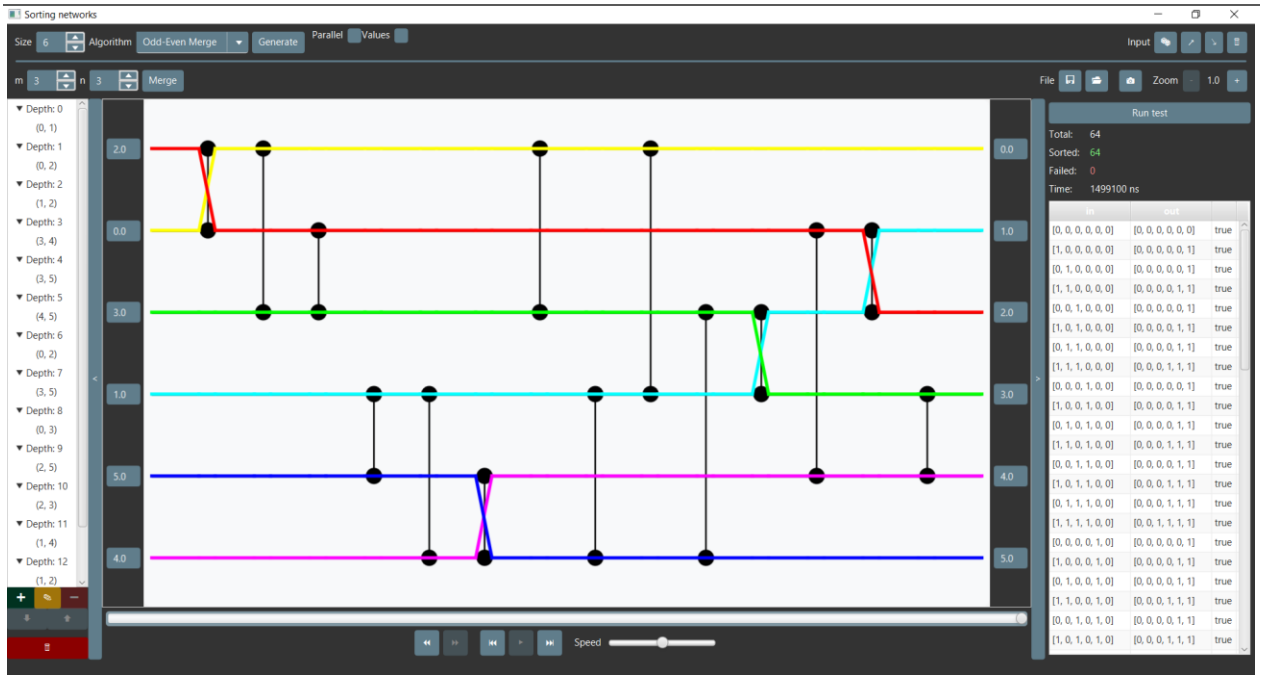
- drawNetwork(network, depths) – atunci când nu exista date de intrare
- drawComputation(network, depths, input) – atunci când exista date de intrare

Funcția drawComputation șterge desenul existent, apelează drawNetwork() și apoi începe desenarea calculului. Prima dată construiește o lista de obiecte WireTrail, o clasa ce ajuta la memorarea drumului parcurs de valorile de intrare. După acest pas se va itera prin fiecare adâncime și apoi prin fiecare linie de date și se va desena segmentul de culoare corespunzătoare și o interschimbare acolo unde este necesar.

2.4 CONCLUZIILE CAPITOLULUI

In acest capitol au fost prezentate tehnologiile utilizate, JavaFX și SceneBuilder pentru dezvoltare și Junit pentru testare, arhitectura generala a aplicatiei, impartirea pe pachete și relatiile dintre clasele mari, și diversii algortimi implementati, Batcher odd-even merge sort fiind cel mai important.

3 DESCRIEREA APLICAȚIEI



Interfața aplicației poate fi împărțită în cinci zone, în stânga meniul de editare, în dreapta meniul de testare, în mijloc reprezentarea grafică și datele de intrare și ieșire ale rețelei, bara de jos pentru controlul animației, iar bara de sus conține funcționalități de generare în stânga și diverse alte funcționalități în dreapta.

Implicit meniul de editare și meniul de meniul de testare sunt ascunse.

Meniul de generare de rețele

Conține 3 opțiuni pentru algoritmi: insertion sort, bubble sort și Batcher odd-even mergesort.

Meniul de combinare de rețele

Această funcționalitate va trata rețeaua curentă ca fiind două rețele distincte, una formată din primele m fire, și a doua formată din următoarele n. La apăsarea butonului „Merge”, se vor adaugă comparatorii necesari pentru combinarea celor două rețele menționate. Se folosește algoritmul Batcher odd-even merge, similar cu funcționalitatea de generare, dar sărind primul pas, și anume apelul recursiv pe cele două subrețele.

Meniul de testare a rețelei

Respectând principiul zero-unu, acest meniu oferă posibilitatea testării tuturor secvențelor de 0 și 1, la sfârșitul execuției punând la dispoziția utilizatorilor un tabel cu rezultatele.

Meniul de editare a rețelei

Conține următoarele funcționalități:

- Adăugare de comparatori
- Editarea comparatorului selectat
- Ștergerea comparatorilor selectați
- Mutarea în sus sau în jos a comparatorului selectat
- Golirea rețelei

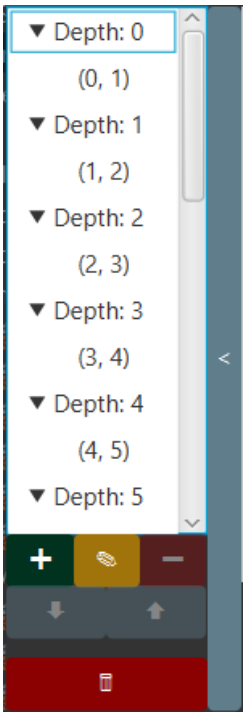


Figura 10 Meniul de editare

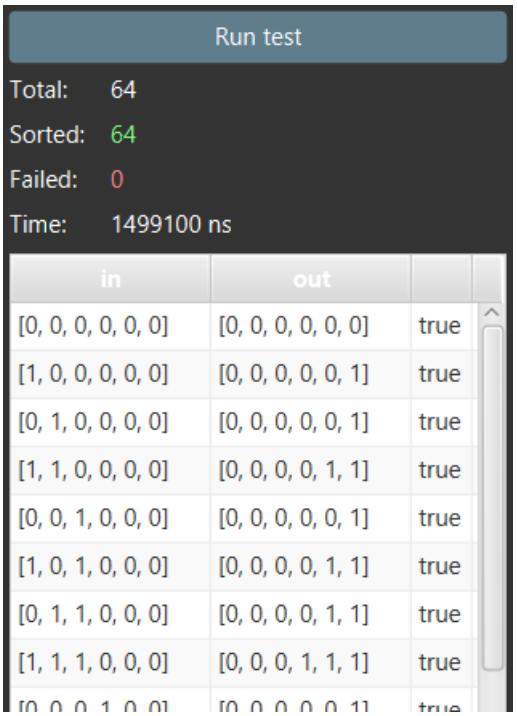
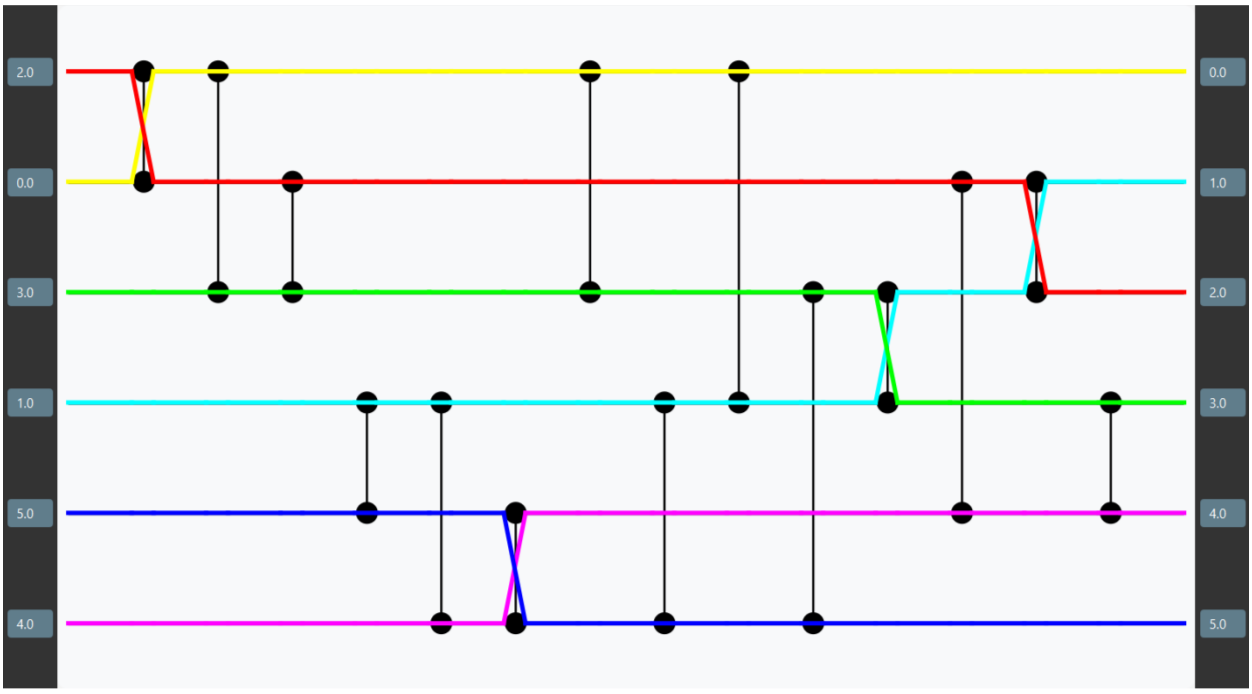


Figura 11 Meniul de testare

Reprezentarea grafică



Câmpurile din stânga reprezintă datele de intrare, iar câmpurile din dreapta datele de ieșire. Firele orizontale reprezintă liniile de date, iar cercurile și liniile orizontale ce le unesc reprezintă comparatorii. Liniile colorate reprezintă drumul urmat de valori pana la poziția potrivita.

4 CONCLUZII FINALE

In aplicația prezentată in această lucrare am dezvoltat un mediu de studiu al rețelelor de sortare, prin combinarea de algoritmi existenți și algoritmi programați de mine am pus la dispoziția utilizatorilor multe funcționalități utile lucrului cu aceste rețele. Limbajul de programare Java a făcut implementarea diverșilor algoritmi foarte ușoara, iar mediul de dezvoltare oferit de JavaFX și SceneBuilder a facilitat construcția aplicației.

Pe viitor, pentru extinderea funcționalităților existente se pot adaugă mai mulți algoritmi de generare și mai multe filtre, pentru îmbunătățirea performantei ar trebui împărțita construirea reprezentării grafice în bucăți și executarea acestora asincron, iar pentru adăugarea de funcționalități noi, deocamdată singura interacțiune oferită de desenul rețelei este selectarea de comparatori, momentan rămânând neimplementate adăugarea, editarea și mutarea.

BIBLIOGRAFIE

1. Knuth, D. E. (1997). The Art of Computer Programming, Volume 3: Sorting and Searching
2. Parberry, Ian (1991). “A Computer Assisted Optimal Depth Lower Bound for Nine-Input Sorting Networks”. Mathematical Systems Theory
3. Bose, R. C.; Nelson, R.J. (1962). “A sorting problem”. Journal of the ACM Volume 9
4. Notices of the American Mathematical Society (1967) pg. 283
5. Batcher, K. E. (1968). “Sorting networks and their applications”. Proc. AFIPS Spring Joint Computer Conference. pg. 307-314
6. Codish, Michael; Cruz-Filipe, Luís; Frank, Michael; Schneider-Kamp, Peter (2014). Twenty-Five Comparators is Optimal when Sorting Nine Inputs (and Twenty-Nine for Ten)
7. Bundala, D.; Závodný, J. (2014). Optimal Sorting Networks. Language and Automata Theory and Applications. Lecture Notes în Computer Science
8. Ajtai, M.; Komlós, J.; Szemerédi, E. (1983). An O(n log n) sorting network
9. Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L. (1990). Introduction to Algorithms
10. "1.2 Design Goals of the Java™ Programming Language". Oracle
11. https://worldwide.espacenet.com/patent/search/family/024573610/publication/US3029413A
12. https://github.com/jix/sortnetopt
13. https://octoverse.github.com/#top-languages
14. https://pypl.github.io/PYPL.html
15. https://developer.nvidia.com/gpugems/GPUGems2/gpugems2_chapter46.html
16. http://fxexperience.com/wp-content/uploads/2011/08/Introducing-FXML.pdf
17. https://www.w3.org/TR/REC-xml/