**Univerziteti u Novom Sadu**

**Tehnički fakultet „Mihajlo Pupin“**

**Zrenjanin**

**Seminarski rad**

**Predmet: Sistemi veštačke inteligencije**

**Softver za simulaciju A\* algoritma za pretragu**

Profesor: Prof. dr Ivana Berković

Asistent: dr Višnja Ognjenović

Student: Jan Hrćan IT 3/14

Zrenjanin, 2017. godina

**Sadržaj**

[Uvod 3](#_Toc484474107)

[1. A\* Algoritam 4](#_Toc484474108)

[1.1. Nastanak 4](#_Toc484474109)

[1.2. Kratak opis načina rada A\* algoritma 4](#_Toc484474110)

[1.3. Pseudo kod A\* algoritma 4](#_Toc484474111)

[2. Primena algoritma na softver 6](#_Toc484474112)

[2.1. Softver – Primenjen A\* Algoritam 6](#_Toc484474113)

[2.2. Kratak opis korišćene tehnologije 6](#_Toc484474114)

[2.3. Klasa AStar() 6](#_Toc484474115)

[3. Izgled aplikacije i upotreba 10](#_Toc484474116)

[4. Zaključak 14](#_Toc484474117)

[5. Literatura 15](#_Toc484474118)

# Uvod

U ovom seminarskom radu se najpre generalno u skraćenoj formi opisuje sam A\* algoritam, kako je nastao i kako funkcioniše, zajedno sa primerom pseudo koda, pa se dalje opisuje glavna klasa u softveru napravljenog u okviru seminarskog rada koji je pisan na Java programskom jeziku uz upotrebu JavaFX tehnologije oko grafičkih elemenata.

# A\* Algoritam

## Nastanak

A\* algoritam je prvi put bio opisan 1968. godine od strane trojice ljudi na Univerzitetu u Stanfordu. To troje ljudi su Piter Hart, Nils Nilson i Bertram Rafael. Te iste godine, Nilson, koji se tada bavio veštačkom inteligencijom, je hteo da unapredi planiranje puta za malog robota koji se zvao „Shakey the Robot“, i to je bio prototip robota koji je mogao da se kreće kroz sobu i izbegava prepreke. U to vreme, ovaj robot se koristio algoritmom A1 koji je navodno bio bolji od, do tad najpoznatijeg, Djikstrinog algoritma. Dalje, na predlog Pitera Hart-a, definisan je algoritam A2, koji su dalje prethodno pomenuta dvojica, zajedno sa Bertramom Rafaelom usavršili. Posle ovoga se dalje razvio algoritam i sada ga znamo kao A\* algoritam.

## Kratak opis načina rada A\* algoritma

Glavne opisne karakteristike A\* algoritma se mogu svesti na sledeće. Dakle, sistem rada algoritma je takav, da on poseduje nekoliko listi sa kojima operiše. Najpre imamo takozvanu „zatvorenu listu“, u kojoj se beleže čvorovi koji su već ispitani, dalje imamo takozvanu „frindž listu“ ili „otvorenu listu“ koja beleži čvorove koje su u direktnoj vezi sa čvorovima koji su već bili ispitani. Na kraju imamo i izračunavanja udaljenosti od početne tačke do krajnje tačke zajedno sa pred „otprilike“ izračunatim udaljenostima (heuristika).

Izračunavanje heuristike za svaki čvor se radi na sledeći način. Imamo formulu:

f(n) = g(n) + h(n)

u ovoj formuli g(n) predstavlja udaljenost ili put od početne tačke do neke tačke n, dok h(n) predstavlja estimaciju udaljenosti neke tačke n do krajnje tačke.

## Pseudo kod A\* algoritma

Dakle, kao što je pre pomenuto, imamo dve glavne liste, a to je **OTVORENA** lista i **ZATVORENA** lista.

**Pseudo kod:**

**function A\*(start, goal)**

**closedSet := {}**

**openSet := {start}**

**cameFrom := the empty map**

**gScore := map with default value of Infinity**

**gScore[start] := 0**

**fScore := map with default value of Infinity**

**fScore[start] := heuristic\_cost\_estimate(start, goal)**

**while openSet is not empty**

**current := the node in openSet having the lowest fScore[] value**

**if current = goal**

**return reconstruct\_path(cameFrom, current)**

**openSet.Remove(current)**

**closedSet.Add(current)**

**for each neighbor of current**

**if neighbor in closedSet**

**continue**

**tentative\_gScore := gScore[current] + dist\_between(current, neighbor)**

**if neighbor not in openSet**

**openSet.Add(neighbor)**

**else if tentative\_gScore >= gScore[neighbor]**

**continue**

**cameFrom[neighbor] := current**

**gScore[neighbor] := tentative\_gScore**

**fScore[neighbor] := gScore[neighbor] + heuristic\_cost\_estimate(neighbor, goal)**

**return failure**

**function reconstruct\_path(cameFrom, current)**

**total\_path := [current]**

**while current in cameFrom.Keys:**

**current := cameFrom[current]**

**total\_path.append(current)**

**return total\_path**

# Primena algoritma na softver

## Softver – Primenjen A\* Algoritam

Softver koji je napravljen u okviru ovog seminarskog rada se zove AppliedAStarAlgorithm(v2.0), i to je zapravo aplikacija koja kroz definisanje početne i krajnje tačke, kao i tačaka koje su blokirane, ili takozvane „prepreke“, rešava problem tako što nalazi najkraći mogući put od početne do krajnje tačke.

## Kratak opis korišćene tehnologije

Ovaj softver je pisan u programskom jeziku Java, koristeći JavaFX tehnologiju koja manipuliše sa grafičkim elementima u modernim Java aplikacijama i apletima. Sam pregled koda funkcionisanja JavaFX tehnologije neće biti izlagan u ovom radu, niti rad sa grafičkim elementima, isključivo objašnjenje i opis koda za klasu AStar.

## Klasa AStar()

public class AStar

je klasa definisana u okviru AppliedAStarAlgorithm(v2.0).

*/\*\* cost fields \*/*private final int DIAGONAL\_TRAVEL\_COST = 14; // hypotenuse in a triangle is 1.4 (multiplied by 10)  
private final int HORIZONTAL\_VERTICAL\_TRAVEL\_COST = 10;  
  
*/\*\* A\* algorithm fields\*/*private PriorityQueue<NodeCell> opened;  
private NodeCell[][] allCells;  
private NodeCell[][] closed;  
private NodeCell start;  
private NodeCell end;  
private static Integer *sizeX*;  
private static Integer *sizeY*;

Imamo dve finalne promenljive (vrednost im se ne može menjati) koje nam predstavljaju g(n) moguće vrednosti iz prethodno navedene formule i kao što je objašnjeno u komentaru, razlozi zbog kojih je dijagonalna cena 14 a vertikalna i horizontalna 10 jeste što hipotenuza u trouglu gde su a i b jednake 1, c, ili hipotenuza će biti jednaka 1,4 pa kada sve pomožimo sa 10, dobijamo cenu puta.

Takođe imamo i liste, dakle OTVORENA lista, koja je zapravo PriorityQueue (red sa prioritetom (vrednosti su poređane u listu od najmanjeg ka najvećem)), što znači da to odgovara algoritmu da iz otvorene liste uvek vadi najmanju vrednost. Dalje imamo zatvorenu listu, početnu i krajnju tačku i veličinu same „rešetke“ tj. grafa.

*/\*\* The main algorithm method \*/*private void algorithm(){  
 assignStartAndEnd();  
 calculateHeuristicsUsingManhattan(); // calculating random distance from end node  
 setBlockedCells(); // setting the blocked cells to null in the list  
 this.start.setF\_value(0); // the starting node has 0 on the F value  
 this.opened.add(this.allCells[start.getI()][start.getJ()]); // starting cell has been added into the queue  
  
 NodeCell node; // node used for examination  
 while(true) {  
 node = this.opened.poll(); // takes out the head value from the queue  
 if (node == null) break; // poll() returns null if the queue is empty so it needs to be checked  
 this.closed[node.getI()][node.getJ()] = node; // the node is put into the closed list  
 if (node.equals(this.end)) return; // if the algorithm has reached the end node while loop is terminated  
  
 NodeCell node\_t; // node used for comparing it to the main node  
  
  
 if (node.getI()-1 >= 0){  
 // |\*|c|\*|  
 // |\*| |\*|  
 // |\*|\*|\*|  
 node\_t = allCells[node.getI()-1][node.getJ()];  
 check(node\_t, node, HORIZONTAL\_VERTICAL\_TRAVEL\_COST+node.getF\_value());  
 if (node.getJ()-1 >= 0){  
 // |c|\*|\*|  
 // |\*| |\*|  
 // |\*|\*|\*|  
 node\_t = allCells[node.getI()-1][node.getJ()-1];  
 check(node\_t, node, DIAGONAL\_TRAVEL\_COST+node.getF\_value());  
 }  
 if (node.getJ()+1 < allCells[0].length){  
 // |\*|\*|c|  
 // |\*| |\*|  
 // |\*|\*|\*|  
 node\_t = allCells[node.getI()-1][node.getJ()+1];  
 check(node\_t, node, DIAGONAL\_TRAVEL\_COST+node.getF\_value());  
 }  
 }  
 if (node.getI()+1 < allCells.length){  
 // |\*|\*|\*|  
 // |\*| |\*|  
 // |\*|c|\*|  
 node\_t = allCells[node.getI()+1][node.getJ()];  
 check(node\_t, node, HORIZONTAL\_VERTICAL\_TRAVEL\_COST+node.getF\_value());  
 if (node.getJ()-1 >= 0){  
 // |\*|\*|\*|  
 // |\*| |\*|  
 // |c|\*|\*|  
 node\_t = allCells[node.getI()+1][node.getJ()-1];  
 check(node\_t, node, DIAGONAL\_TRAVEL\_COST+node.getF\_value());  
 }  
 if (node.getJ()+1 < allCells[0].length){  
 // |\*|\*|\*|  
 // |\*| |\*|  
 // |\*|\*|c|  
 node\_t = allCells[node.getI()+1][node.getJ()+1];  
 check(node\_t, node, DIAGONAL\_TRAVEL\_COST+node.getF\_value());  
 }  
 }  
 if (node.getJ()-1 >= 0){  
 // |\*|\*|\*|  
 // |c| |\*|  
 // |\*|\*|\*|  
 node\_t = allCells[node.getI()][node.getJ()-1];  
 check(node\_t, node, HORIZONTAL\_VERTICAL\_TRAVEL\_COST+node.getF\_value());  
 }  
 if (node.getJ()+1 < allCells[0].length){  
 // |\*|\*|\*|  
 // |\*| |c|  
 // |\*|\*|\*|  
 node\_t = allCells[node.getI()][node.getJ()+1];  
 check(node\_t, node, HORIZONTAL\_VERTICAL\_TRAVEL\_COST+node.getF\_value());  
 }  
  
 }  
}

U ovoj metodi, pod nazivom *algorithm()* se izvršavaju neke unutrašnje metode, od kojih je prva *assignStartAndEnd():*

*/\*\* assigning start and end \*/*private void assignStartAndEnd (){  
 for (int i=0; i<*sizeX*; ++i){  
 for (int j=0; j<*sizeY*; ++j){  
 if (allCells[i][j].isStart()){  
 this.start = allCells[i][j]; // setting the starting cell  
 }  
 else if (allCells[i][j].isEnd()){  
 this.end = allCells[i][j]; // setting the ending cell  
 }  
 }  
 }  
}

Dakle, kao što vidimo, ova metoda postavlja vrednosti promenljivih *start* i *end* koje se dalje koriste u izvršavanju algoritma kao početna i krajnja tačka.

Dalje se poziva metoda pod nazivom *calculateHeuristicsUsingManhattan():*

*/\*\* Method for calculating heuristics  
 \* (calculating the approximate distance from the end node)\*/*private void calculateHeuristicsUsingManhattan (){  
 if (this.allCells != null){  
 for (int i=0; i<*sizeX*; i++){  
 for (int j=0; j<*sizeY*; j++){  
 this.allCells[i][j].setH\_value((this.end.getI()- i)+(this.end.getJ()-j));  
 }  
 }  
 }  
}

Računanje h(n) vrednosti iz formule se najčešće koristi metod koji se zove Menhetn, koji je dobio naziv po ostrvu Menhetn, Nju Jork, zbog specifičnog rasporeda ulica i avenija. U kodu se može videti način na koji se izračunava h(n) za svaku ćeliju posebno.

Na kraju se poziva i treća metoda pod nazivoma *setBlockedCells():*

*/\*\* Method for setting blocked cells  
 \* (if a cell is blocked it gets a NULL value  
 \* in the allCells list)\*/*private void setBlockedCells(){  
 if (this.allCells != null){  
 for (int i=0; i<*sizeX*; i++){  
 for (int j=0; j<*sizeY*; j++){  
 if (this.allCells[i][j].isBlocked()){  
 this.allCells[i][j] = null;  
 }  
 }  
 }  
 }  
}

Dakle, kao što se može videti, ova metoda pronalazi sve označene „zablokirane“ ćelije i postavlja njihovu vrednost kao NULL u glavnoj listi svih čvorova kako bi glavni algoritam mogao da prepozna da li sme koristiti tu ćeliju ili ne.

Dalji koraci u metodi *algorithm()* su generalno kroz komentare objašnjeni, i kao što vidite u daljim koracima se za svaku ćeliju proveravaju susedne ćelije i za svaku ćeliju se poziva metoda pod nazivom *check():*

*/\*\* Method for updating the cost  
 \* and parenting and re-parenting nodes\*/*private void check(NodeCell \_node\_t, NodeCell \_node, int \_cost){  
 if (\_node\_t == null || this.closed[\_node\_t.getI()][\_node\_t.getJ()] == \_node\_t) return; // if the node is blocked or closed just return  
  
 int \_node\_t\_tempFValue = \_node\_t.getH\_value() + \_cost; // temp value used for re-parenting if needed  
  
 if (\_node\_t.getF\_value()>\_node\_t\_tempFValue || !opened.contains(\_node\_t)){  
 \_node\_t.setF\_value(\_node\_t\_tempFValue);  
 \_node\_t.setParent(\_node);  
 if (!opened.contains(\_node\_t)) opened.add(\_node\_t);  
 }  
}

Kao što se može videti iz priloženog koda, ova metoda uzima argumente čvora koji se proverava, čvora koji je roditelj tom čvoru koji se proverava, i sama cena puta. Krajnji uslov jeste, jedna jako bitna stvar, da ako neki čvor već ima cenu, ali sada nova cena je „jeftinija“ od prethodne, moramo da odradimo takozvanu „promenu roditelja“ kako bismo dobili najoptimalniju finalnu cenu, takozvani najoptimalniji f(n).

Na kraju nam je jedino ostala jedina javna metoda AStar klase pod nazivom *doAlgorithm():*

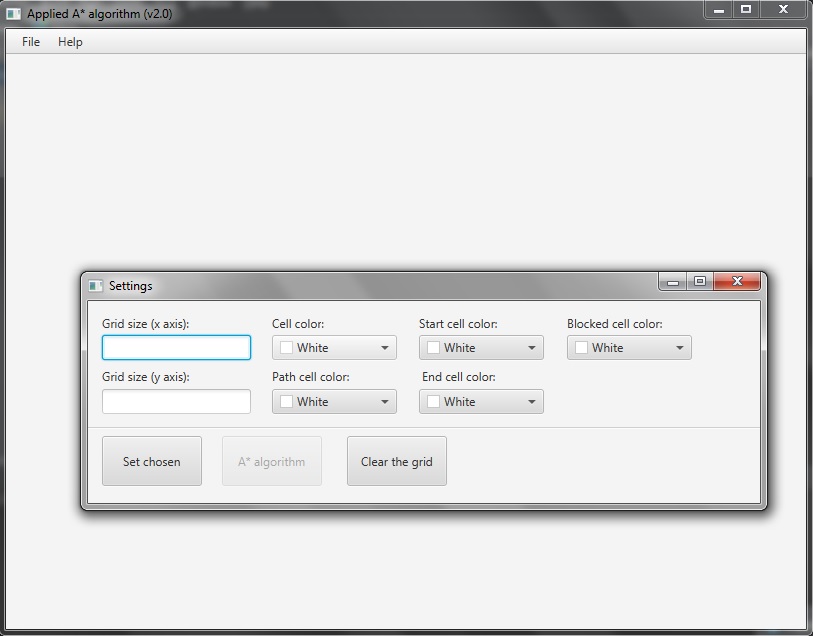
*/\*\* main public method that starts the algorithm and returns data\*/*public List<NodeCell> doAlgorithm(){  
 algorithm(); // calls the main algorithm method  
  
 List<NodeCell> list = new ArrayList<NodeCell>();  
 if (this.closed[end.getI()][end.getJ()] == end){  
 NodeCell node = allCells[end.getI()][end.getJ()];  
 while(node.getParent() != null){  
 list.add(node.getParent());  
 node = node.getParent();  
 }  
 }  
 if (!list.isEmpty()) list.remove(list.size()-1);  
 return list;  
}

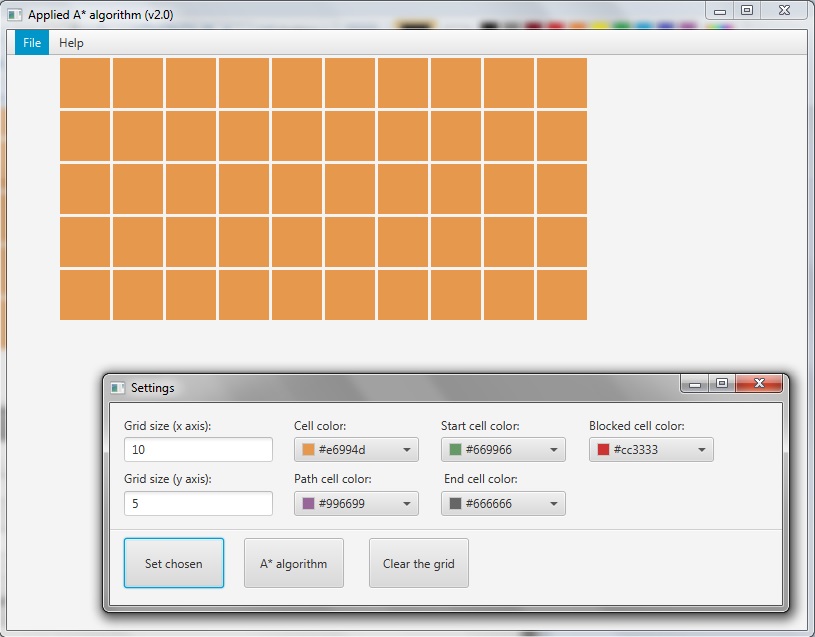
Ova javna metoda poziva privatnu metodu *algorithm()* i vraća listu čvorova koji po algoritmu spadaju u najkraći mogući put od početne do krajnje tačke.

# Izgled aplikacije i upotreba

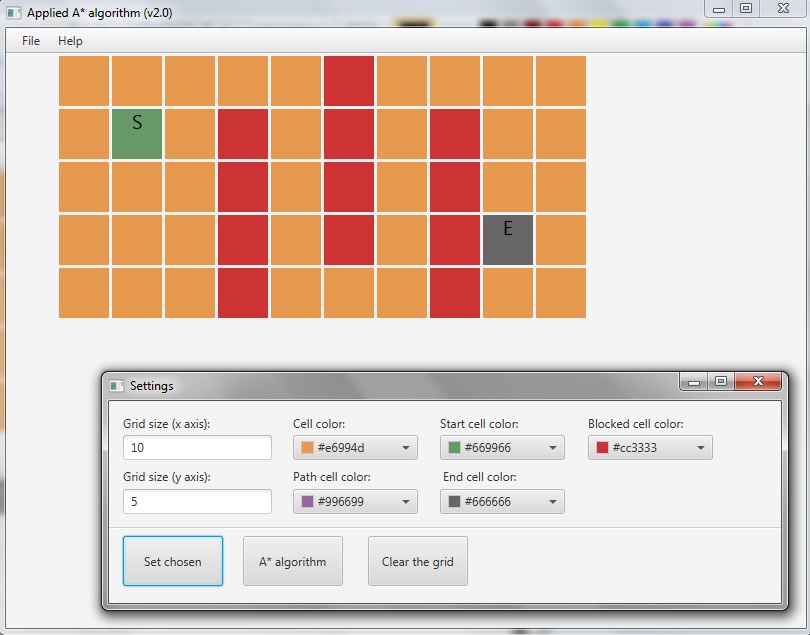
Dakle, kroz naredne slike se može videti upotreba i sam izgled aplikacije:

Glavni prozor i prozor sa podešavanjima:

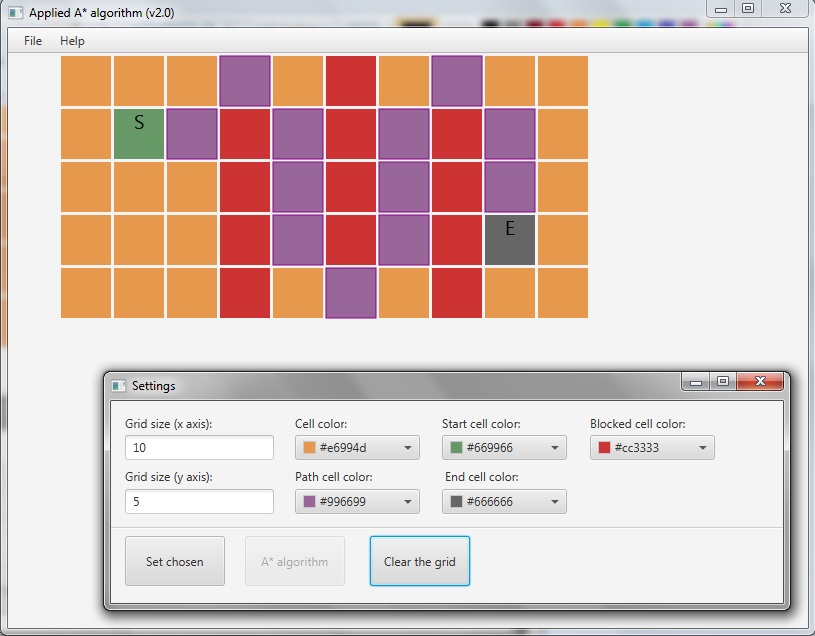


Nameštena podešavanja i napravljena rešetka tj. graf na kome će se izvršavati algoritam:

Postavljene tačke koje su kraj i početak i takođe postavljene ćelije (čvorovi) koji su „zablokirani“:



Izvršen algoritam na dugme A\* algorithm:



# Zaključak

Generalno, A\* jeste trenutno najkorišćeniji algoritam za pretragu najkraćeg i najboljeg puta na grafu i koristi se široko za rešavanje raznih aspekata programiranja i raznih problema vezanih za pronalaženje najkraćeg puta. U radu je ukratko, kako o algoritmu tako i o softveru napravljenog u okviru seminarskog rada, objašnjeno osnovno funkcionisanje i implementacija, kako samog algoritma, tako i softvera.

# Literatura

<http://web.mit.edu/eranki/www/tutorials/search/>

[https://en.wikipedia.org/wiki/A\*\_search\_algorithm#History](https://en.wikipedia.org/wiki/A*_search_algorithm#History)

<http://mnemstudio.org/path-finding-a-star.htm>

<http://mat.uab.cat/~alseda/MasterOpt/AStar-Algorithm.pdf>

<http://coecsl.ece.illinois.edu/ge423/lecturenotes/AstarHandOut.pdf>

<http://db.cs.duke.edu/courses/fall11/cps149s/notes/a_star.pdf>