Robin Rajšić

Developed in C++

MAZe RUNNER

Algorithm

Contents

[Uvod 2](#_Toc34567896)

[Znakovne oznake elemenata i dizajni labirinta 3](#_Toc34567897)

[Dizajn 1 3](#_Toc34567898)

[Dizajn 2 4](#_Toc34567899)

[Dizajn 3 4](#_Toc34567900)

[Kretanje 4](#_Toc34567901)

[Algoritam 5](#_Toc34567902)

[Varijable 5](#_Toc34567903)

[Metode 5](#_Toc34567904)

[Rješenje labirinta 3 9](#_Toc34567905)

[Rješenje labirinta 2 10](#_Toc34567906)

[Rješenje labirinta 1 11](#_Toc34567907)

[Rezultati 12](#_Toc34567908)

[Zaključak 13](#_Toc34567909)

# Uvod

U ovom seminaru ćemo detaljno analizirati jednostavan algoritam za prolazak puta kroz labirint. Testirati ćemo ga sa tri labirinta različite veličine. Promotriti ćemo ukupan broj koraka u odnosu na sa brojem koraka potrebnih da bi se prošlo kroz dani labirint te (otprilike) izračunati efektivnost i razmisliti o mogućim poboljšanjima algoritma.

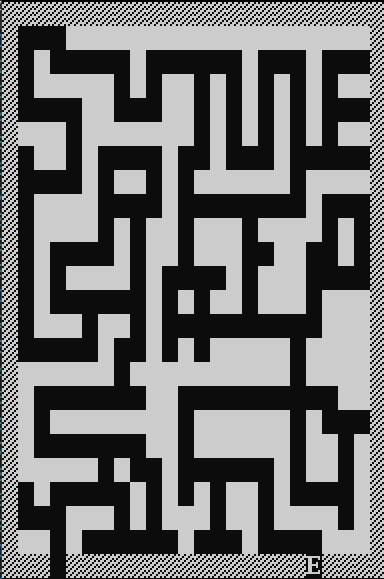
# Znakovne oznake elemenata i dizajni labirinta

Da bi postigli što veću moguću jednostavnost i fokusirali se samo na algoritam, cijeli labirint je dvodimenzionalno polje znakova u konzoli. Svaki element je jedan znak. Koristit ćemo se proširenom ASCII tablicom.

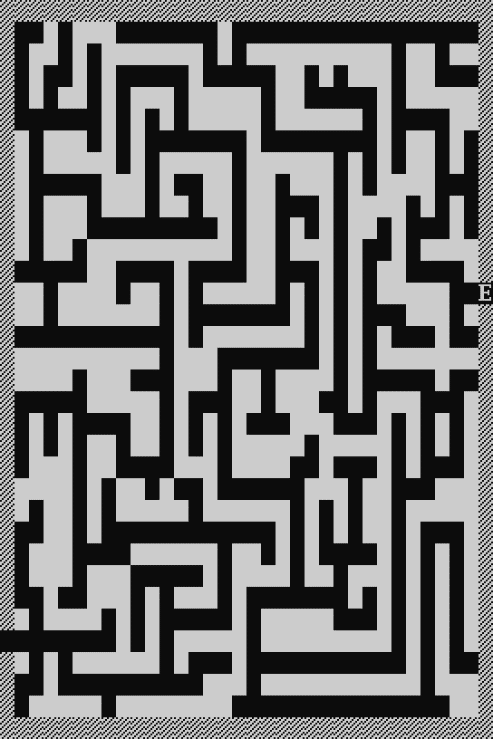
* Igrač - 
* Vanjski zid - 
* Unutarnji zid - 
* Prijeđen put - 
* Izlaz - 

Ulazak je prazno mjesto, kao i put.

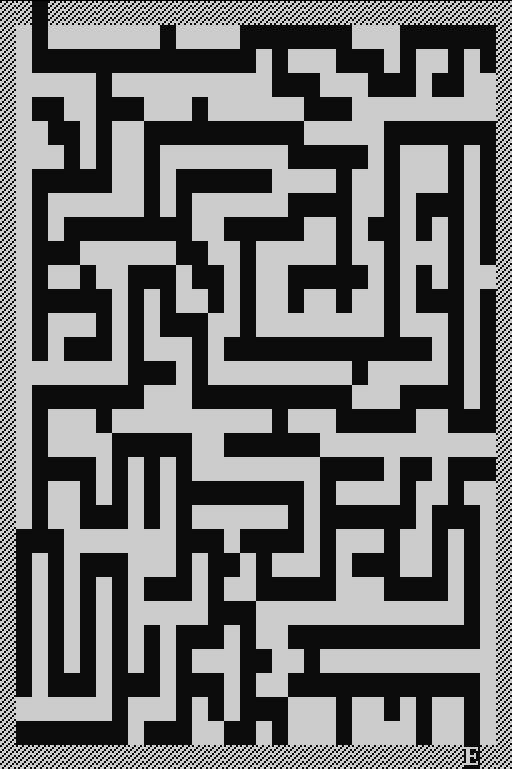
## Dizajn 1

****

## Dizajn 2



## Dizajn 3



Kretanje – „igrač“ će se kretati po labirintu sa potezima koji vrijede za dvodimenzionalnu matricu. Potez gore je (row – 1)(col), potez dolje (row + 1)(col), potez desno (row)(col + 1) i potez lijevo (row)(col – 1).

# Algoritam

**Klasa Maze**

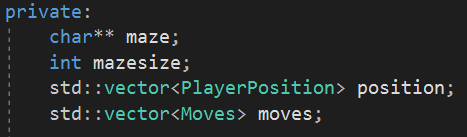
U suštini cijeli program se nalazi u metodama klase Maze. Fokusirati ćemo se metode koje su bitne za ovaj seminar te ih detaljno analizirati, a ostale metode poput dizajna i ispisa na ekran ćemo samo ukratko opisati.

**Klasa PlayerPosition** – Klasa koja ima dvije varijable (int row i int col) koje ćemo koristiti za poziciju unutar labirinta. Row će označavati broj retka, a col broj stupca.

**Enumerator Moves** - Pomoću nje dajemo nazive svakom „potezu“ igrača te olakšavamo čitljivost koda. Potezi su: UP, DOWN, RIGHT, LEFT.

**Struktura Character -** Sadrži oznake elemenata. Povećava čitljivost i uklanjamo magične brojeve.

## Varijable



-maze je dvodimenzionalno polje znakova, mazesize je varijabla koju koristimo sa dizajnom labirinta (naši dizajni su: 24x24, 32x32, 34x34).

-Stog varijabli position i moves su tipa std::vektor koji se u C++-u koristi kao stog sa sljedećim funkcijama koje ćemo koristiti:

* 1. back() – vraća poslijednji objekt sa stoga
  2. push\_back(objekt) – stavlja objekt u stog
  3. pop\_back() – izbacuje zadnji objekt sa stoga
  4. size() – vraća broj objekata u stogu

## Metode

void mazeDesign\_1, mazeDesign\_2 i mazeDesign\_3 su metode u kojima popunjavamo labirinit sa elementima.

void end() je metoda u kojoj brišemo dinamički alociranu memoriju.

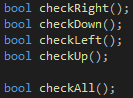
 - Pronalazi i vraća poziciju ulaza u labirint

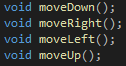
 - Pronalazi i vraća poziciju izlaza iz labirinta

 - Postavlja poziciju ulaza kao prvu varijablu u stog position.

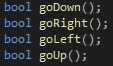
 - markPlayer postavlja oznaku igrača () na trenutnu poziciju u stog position, a markPath postavlja oznaku prijeđenog puta () na trenutnu poziciju.

Sljedeće metode su repetitivne i mjenja se samo smjer, pa je dosta opisati jednu metodu da bi se razumijele sve.

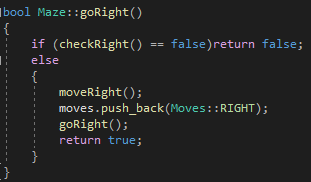
checkRight provjerava da li je za jedno mjesto u desno slobodan put, te ako je onda vraća true, ako je zid onda vraća false. Isto vrijedi I za druge smjerove.

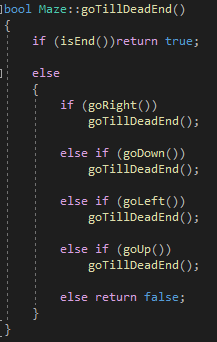
checkAll pomoću svih check metodi provjerava da li je u bilo kojem smjeru slobodan put (ako je u jednom smjeru označen prijeđen put  onda vraća false kao da je i zid.

moveDown postavlja oznaku prijeđen put  na trenutnu poziciju, dodaje novu poziciju za jedno mjesto prema dolje u stog position i označuje igrača na toj novoj poziciji. Pozicija se dodaje prema pravilima opisanim prethodno u **kretanju.**

Sljedeće funkcije su rekurzivne.

Ponovo ću se bazirati samo na jednoj jer isto vrijedi i za sve ostale.

 Kao izlaznu točku imamo uvijet checkRight(). Tek kada taj uvjet vrati false (kada je desno zid) rekurzija je prekinuta. Ako uvjet nije ispunjen onda se pomakni u desno, dodaj potez u vektor poteza i pozovi samu sebe.

S ovom funkcijom će se „igrač“ kretati sve dok ne dođe do slijepe ulice.

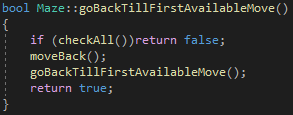
Postoje dva izlazna stanja.

Prvi je uvjet isEnd() ( isEnd() vraća true ako je igrač došao do kraja labirinta).

Ako može ići u bilo koju stranu onda će pozivati samu sebe dok ne dođe do “slijepe ulice”, onda vraća stanje false.

Na slici ispod vidimo gdje algoritam završi ako se pozove samo ta funkcija.

Zato nam je potrebna još jedna rekurzivna funkcija, koja će vračati igrača do prvog „slobodnog“ raskrižja.



moveBack() je funkcija koja vraća igrača za jedan korak u nazad na sljedeći način: uzima poslijednji potez iz stoga „moves“ i izbacuje ga zove funkciju za pomicanje u suprotnu stranu. Npr, ako je poslijednji potez bio prema dolje onda će se izbaciti DOWN i igrač će se pomaknuti prema gore (UP).

Izlazna točka nam je funkcija checkAll() (koja vraća true ako se igrač može pomaknuti u bilo koju stranu).

Do tada se pomiče za korak u nazad i zove samu sebe

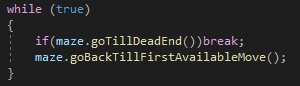
Ako implementiramo i ovu funkciju u algoritam dobijemo sljedeći rezultat:



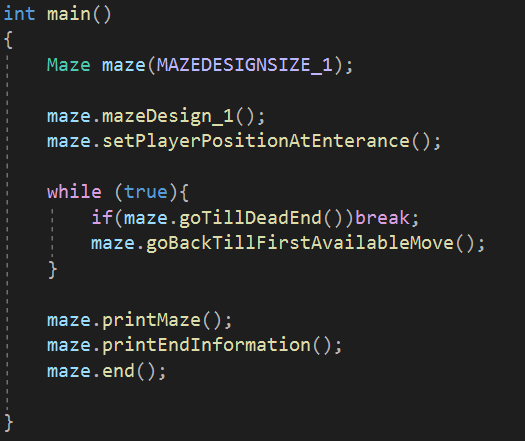
Ako ponovo pozovemo funkciju

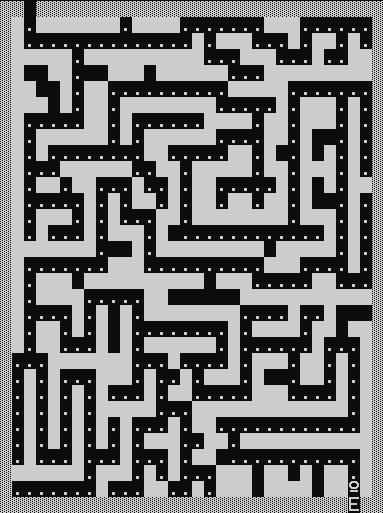


Sada vidimo da samo naizmjeničnim pozivanjem ovih dviju funkcija algoritam će pronaći put kroz labirint.

Podsjetimo se da funkcija goTillDeadEnd() vraća true ako se ispuni uvjet isEnd() koji vraća true ako je igrač došao do kraja labirinta.

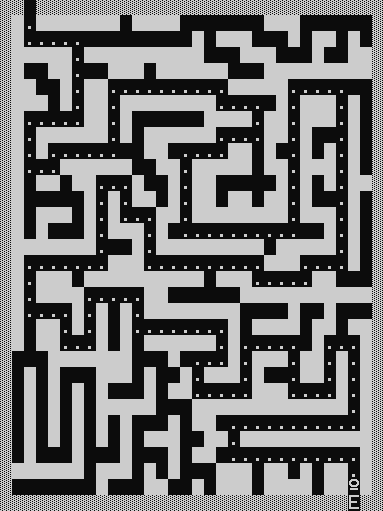
I naposlijetku ovako izgleda cijeli program:



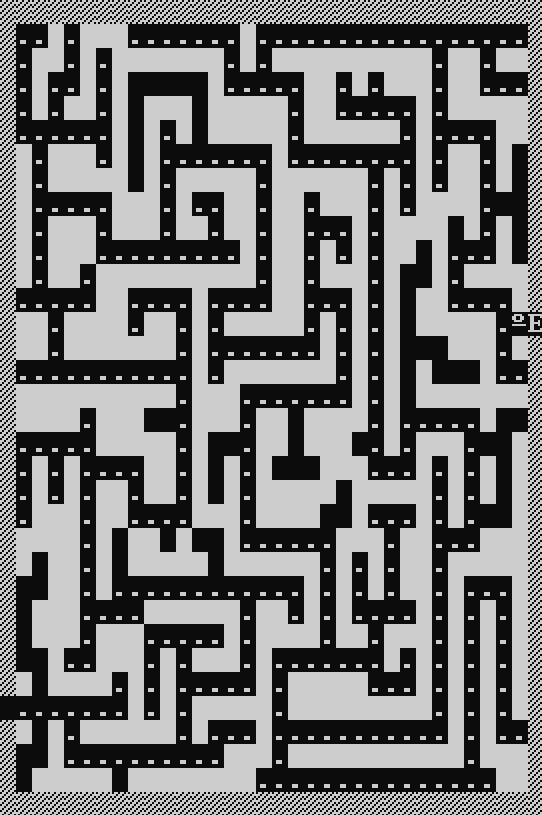
Rješenje labirinta 3

Kao što vidimo postoje „ulice“ u koje igrač nije išao to je zato što se nalaze usred ravnog puta.

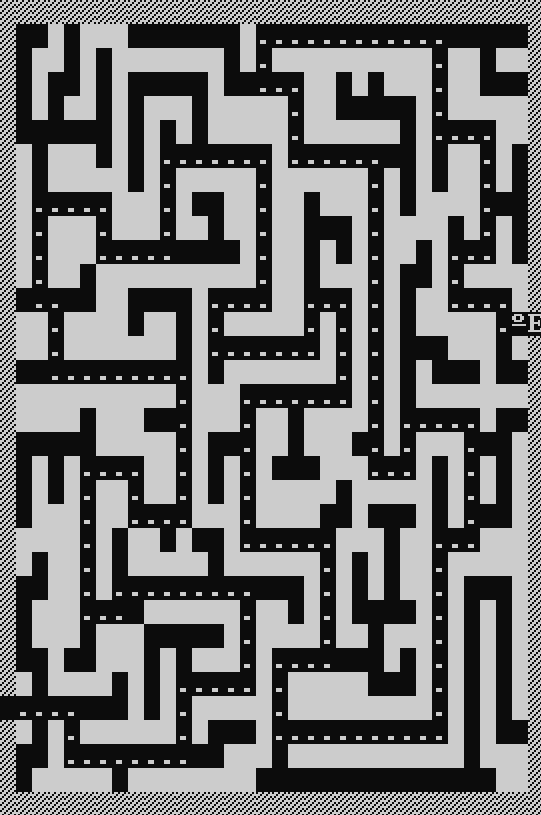
Rješenje bez krivih puteva:



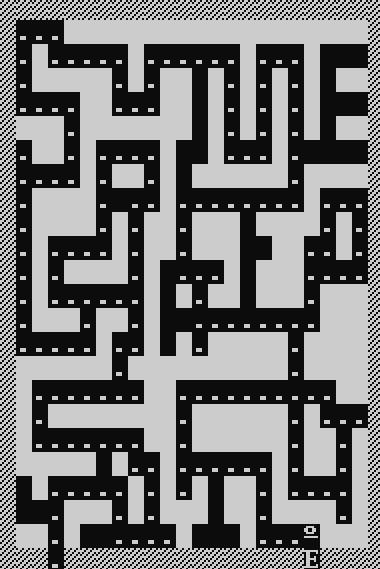
# Rješenje labirinta 2



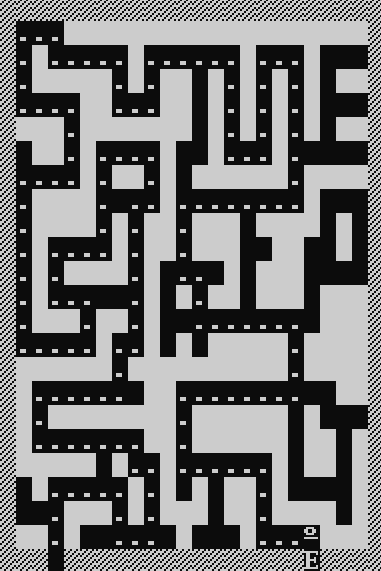
Bez krivih puteva:



# Rješenje labirinta 1



Bez krivih puteva:



# Rezultati

Promatrati ćemo broj ukupnih koraka i broj koraka potrebnih za izlazak iz labirinta.

Labirint 3:



Labirint 2:



Labirint 1:



Vidimo da je dizajn 3 puno jednostavniji od dizajna 1 i 2 te ga nećemo ubrajati.

Efektivnost ćemo izračunati djeljenjem potrebnih koraka sa ukupnim brojem koraka.

Dobije se između **36 – 40 %** efektivnosti na prva dva dizajna.

Složenost ovog algoritma jest O(n).

Moguća poboljšanja:

Cilj ovog projekta je napraviti što jednostavniji algoritam koji će pronaći put kroz labirint. Algoritam je ispravan i trebao bi raditi na bilo kojoj veličini i bilo kakvom dizajnu labirinta. Jednostavnost utječe na efektivnost. Efektivnost bi mogli povećati uz dodavanje dodatnih funkcionalnosti (napraviti algoritam malo pametnijim). Jedna od tih funkcionalnosti bi bila da dodamo još jedan stog tipa vektor u koji ćemo spremati „raskrižja“ te bi se računalo „teleportiralo“ na poziciju raskrižja umjesto da se vraća kada dođe do slijepe ulice. S time smanjiti ukupan broj koraka.

Ako pomnije proučimo rješenje labirinta 3 vidimo da postoji i put sa manje koraka nego put koji je računalo pronašlo uz pomoć algoritma, odnosno „prečice“. Ovaj problem bi se mogao riješiti kada bi modificirali algoritam da na raskrižjima daje prednost smjeru gdje se nalazi izlaz (ako je poznato na kojoj strani se nalazi izlaz).

Dizajni su pre mali i pre jednostavni da bi se ove dodatne funkcionalnosti pokazale učinkovitima pa ćemo ih preskočiti.

# Zaključak

Uz pomoć rekurzija uspijeli smo stvoriti jednostavan algoritam sa ~300 linija koda koji može pronaći put kroz bilo koji labirint. S time uočavamo bitnost rekurzivnih funkcija u situacijama gdje se funkcije moraju ponavaljati dok ne odrade neku svrhu , kao što je u ovom projektu bio primjer kretanja u jednu stranu dok se ne naiđe na zid.

Rekurzivne funkcije smanjuju zalihost koda, malo povećavaju kompleksnost ali puno više povećavaju čitljivost i ponovnu upotrebljivost. Smatram da je rekurzija jako bitan koncept za shvatiti jer može divovske probleme smanjiti na par linija.