# Universiteti i Prishtinës “Hasan Prishtina”

## Fakulteti i Inxhinierisë Elektrike dhe Kompjuterike


# Dokumentim teknik i projektit

**Lënda: Dizajni dhe Analiza e Algoritmeve**

## Titulli i projektit: Rubik’s Cube Solver

| **Emri profesorit/Asistentit** | **Emri & mbiemri studentëve / email adresa** | |
|---|---|---|
| Prof. Dr. Avni REXHEPI<br>MSc. Adrian YMERI | 1. Arlind Nimani | arlind.nimani@student.uni-pr.edu |
| | 2. Enis Shallci | enis.shallci@studentet.uni-pr.edu |
| | 3. Erjon Kosumi | erjon.kosumi@student.uni-pr.edu |
| | 4. Olta Pllana | olta.pllana@student.uni-pr.edu |
| | 5. Qëndresa Potoku | qendresa.potoku@student.uni-pr.edu |
| | 6. Venera Plakolli | venera.plakolli@student.uni-pr.edu |
| | 7. Yllka Bicaj | yllka.bicaj@student.uni-pr.edu |
| | 8. Melos Ymeri | melos.ymeri@student.uni-pr.edu |
| | 9. Feride Berisha | feride.berisha2@student.uni-pr.edu |
| | 10. Festina Imeraj | festina.imeraj@student.uni-pr.edu |

Prishtinë, 2023

# Përmbajtja

# Abstrakti

Kubi i Rubikut është një lojë klasike puzzle që ka qenë një sfidë për entuziastët e enigmave për dekada. Zgjidhja e kubit të Rubikut nuk është vetëm një lojë argëtuese, por është gjithashtu një problem i rëndësishëm në shkencën kompjuterike, pasi përfshin kërkimin e një zgjidhjeje në një hapësirë të madhe kërkimi. Në këtë projekt, ne eksplorojmë dy algoritme të ndryshme, Iterative Deepening Depth-First Search dhe A* Algorithm, për të zgjidhur enigmën e kubit të Rubikut.

Iterative Deepening Depth-First Search është një algoritëm klasik i kërkimit që përdor kërkimin depth-first por me një kufi thellësie që rritet në mënyrë të përsëritur. Nga ana tjetër, A* Algorithm është një algoritëm kërkimi që përdor funksione heuristike për të udhëhequr kërkimin drejt gjendjes së qëllimit. Krahasojmë performancën e këtyre dy algoritmeve për sa i përket kompleksitetit të kohës dhe hapësirës dhe tregojmë se algoritmi A* është më i mirë se algoritmi Iterative Deepening Depth-First Search për sa i përket efikasitetit.

Ne implementojmë zgjidhjen e kubit të Rubikut duke përdorur gjuhën e programimit Python. Programi jonë funksionon duke marrë si hyrje një input dhe nxjerr sekuencën më të shkurtër të lëvizjeve të nevojshme për ta zgjidhur atë. Implementimi përfshin një userinterface grafike që i lejon përdoruesit të vizualizojë procesin e zgjidhjes dhe të ndërveprojë me kubin e Rubikut.

Në përgjithësi, projekti jonë ofron një analizë gjithëpërfshirëse të performancës së dy algoritmeve të ndryshme në zgjidhjen e enigmës së kubit të Rubikut dhe ofron një mjet ndërveprues për zgjidhjen e enigmës së kubit të Rubikut.

# I. Hyrje

Kubi i Rubik-ut paraqet një lojë e cila është e njohur me vite madje edhe dekada si një lojë po aq argëtuese edhe sfiduese. Ajo u shpik në vitin 1974 nga skulptori dhe profesori hungarez i arkitekturës Erno Rubik. Kubi i cili përbëhet nga 6 faqe me nga 9 kube te vogla me ngjyra të ndryshme, ku sfida e kësaj loje është që në gjashtë faqet të radhiten kubet (katrorët) me ngjyrë të njëjtë. Kjo lojë e cila kërkon një logjikë dhe përkushtim për t'u zgjidhur është edhe një problem me rëndësi në fushën e shkencave kompjuterike ku trajtohet nga programerët duke krijuar programe të ndryshme me gjuhë të ndryshme programuese.

Zgjidhja e një kubi të Rubikonit ka qenë një problem vërtet i ndërlikuar dhe njerëzit kanë gjetur zgjidhje që janë të shpejta dhe efikase për të zgjidhur këtë problem. Kompjuterët përdoren gjithashtu për të zgjidhur kubin duke përdorur algoritme të ndryshme, duke përfshirë algoritmin e Korf, algoritmin e Thistlethwaite, algoritmin e Kociemba, algoritmin IDFS ose algoritmin IDA*. Këto janë algoritmet mbizotëruese për të zgjidhur këtë problem. Kubi i Rubikonit është shumë kompleks dhe ka një numër të konsiderueshëm mundësish, gjë që e bën atë tepër të sofistikuar për t'u zgjidhur. Duke aplikuar algoritme për t'u marrë me këtë problem, mund të shihet potenciali i përdorimit të kompjuterëve dhe algoritmeve për të trajtuar problemet me këtë nivel kompleksiteti. Prandaj, me anë të këtij punimi kuptojmë që ka potencial për të zgjidhur probleme në fusha të tjera që janë të rëndësishme për shoqërinë tonë.

Duke parë rëndësinë e këtij problemi logjik në shoqërinë tonë dhe në shkencë është e rëndësishme të merremi me këtë çështje dhe krijimin e një programi që qon ne zgjidhjen e këtij problemi. Kështu që ne kemi ndërtuar një program në gjuhën programuese Python duke përdorur dy algoritme: Iterative Deepening Depth-First Search dhe A*.

Algoritmi IDDFS është një variant i algoritmit Depth-First Search (DFS), i cili është një algoritëm rekurziv që eksploron thellësinë e pemës së kërkimit së pari, përpara se të kthehet prapa dhe të eksplorojë degë të tjera. Ky algoritëm kufizon thellësinë e kërkimit në secilin iterim për të shmangur ngecjen në unazë infinite që eksploron zgjidhjen jo optimale. Duke rritur limitin e thellësisë së secilit iterim, algoritmi siguron gjetjen e zgjidhjes më të shpejtë të mundshme.

Sa i përket algoritmit A* funksionon duke gjetur lëvizjet e mundshme duke u bazuar nga gjendja fillestare deri tek ajo përfundimtare duke mbajtur shënime për koston e rrugës në secilën gjendje dhe një kosto të vlerësuar për të arritur gjendjen e synuar nga çdo gjendje. Algoritmi e zgjedh gjendjen e ardhshme duke u bazuar nga shuma e kostos dhe kostoja për të arritur në gjendjen e destinuar e cila njihet si f-value. Algoritmi vazhdon të eksplorojë gjendjen e cila ka f-value më të vogël deri sa të arrihet gjendja e dëshiruar.

# II. Qëllimi i punimit

Qëllimi i këtij punimi është të ofrojë një njohuri të thellë të Kubit të Rubikut, të algoritmeve të ndryshme, avantazheve dhe disavantazheve të tyre, përdorimit të tyre në gjuhë të ndryshme programuese, si dhe implementimit të tyre. Kryesorja e punimit është të kombinojë të gjitha këto të dhëna dhe të prezantojë zgjidhjen më të mirë për Kubin e Rubikut.
Punimi synon të krijojë një kuptim të plotë rreth strukturës, karakteristikave dhe kompleksitetit të Kubit të Rubikut. Do të analizohen algoritmet e njohura si IDA* dhe IDDFS, për t'u njohur me metodat e tyre të zgjidhjes dhe për të vlerësuar përfitimet dhe kufizimet që ofrojnë.
Për më tepër, projekti do të diskutojë kompleksitetin e kohës dhe hapësirës që lidhet me zgjidhjen e Kubit të Rubikut duke përdorur këto algoritme, duke ofruar njohuri mbi efikasitetin dhe kërkesat për burime të secilës qasje.

# III. Pjesa kryesore

## 3.1 Kubi i Rubik-ut

Kubi i Rubikut është një enigmë tredimensionale që u shpik në vitin 1974 nga një skulptor dhe profesor i arkitekturës hungarez i quajtur Erno Rubik.
Ai u bë shpejt një fenomen mbarëbotëror dhe është ende i popullarizuar sot.
Kubi i Rubikut përbëhet nga 27 kube më të vegjël, të vendosur në një rrjet 3x3x3. Secili prej kubeve më të vegjël quhet "cubie".

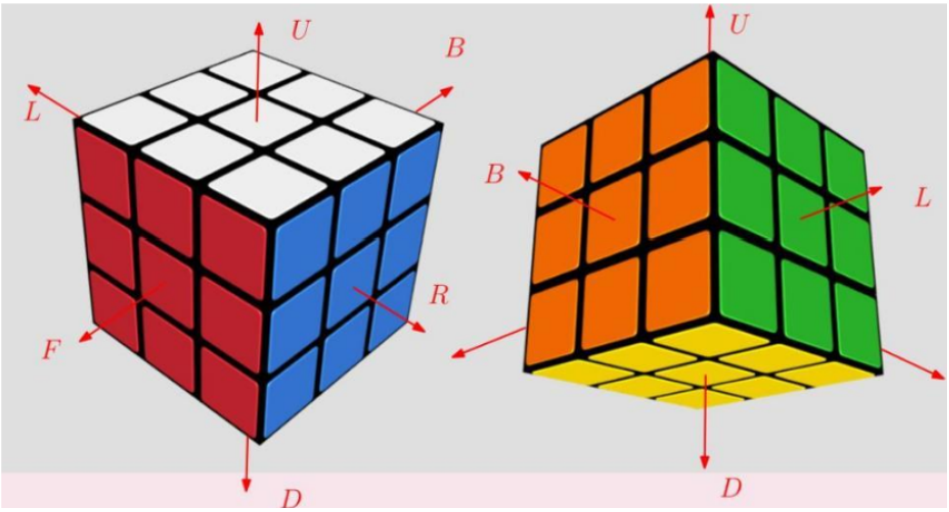


*Figura 1. Kubi i Rubikut*

Kubi i Rubikut ka 8 kënde të cilat mund të renditen në 8! (40,320) permutacione.
Kubat e Rubikut janë enigma të ndërlikuara për t'u zgjidhur me një total prej 43,252,003,274,489,856,000 kombinimesh të mundshme [1].

$$\frac{c_a! \cdot c_c^{c_a-1} \cdot e_a! \cdot e_c^{e_a-1}}{2} = \frac{8! \cdot 3^7 \cdot 12! \cdot 2^{11}}{2}$$

$$c_a = \text{corner amount}$$
$$c_c = \text{corner colours}$$
$$e_a = \text{edge amount}$$
$$e_c = \text{edge colours}$$

*Figura 2. Permutacionet e një ekuacioni të kubit të Rubikut*

## 3.2 Algoritmet

Për të gjetur zgjidhjen e një kubi është e rëndësishme të kuptohet se ky është njw search-problem.Në rastin e kubit të Rubikut, numri i gjendjeve, siç përmendet me lartw, është mjaft i madh.Andaj aplikohen algoritmet, për të minimizuar kohën e kërkimit (search time).
Për të thjeshtuar problemet e avancuara si kubi i Rubikut, është krijuar diçka që quhet state tree.
Një state tree thjesht përmban të gjitha gjendjet në të cilat mund të ekzekutohet algoritmi i kërkimit.States nw pemë quhen nyje, dhe decisions quhen degë që lidhin nyjet.
Për të ulur kohën e kërkimit aplikohen **heuristics**. Formula për një heuristic search është **f = g + h**, ku g është kostoja e përdorur për të arritur current state nga start state, në rastin tonë, është sasia e lëvizjeve të aplikuara dhe h është kostoja e nevojshme për të arritur në solution state, në rastin tonë, është sasia e lëvizjeve të nevojshme për zgjidhjen e kubit [3].

### 3.2.1 Depth first search

Algoritmi DFS do të shkojë në state tree duke parë thellësinë fillimisht. Në fig. 3 mund të shihet një shembull se si DFS do të kërkojë përmes një binary state tree. Ku A është start state. Këtu algoritmi do të zgjedhë nyjen e djathtë deri sa të arrihet fundi i pemës. Në fig. 3 janë paraqitur shigjeta që shkojnë nga A në D. Kur të arrijë në fund të pemës së kërkimit, algoritmi më pas do të rizgjidhë decisionin e fundit që na bën të ndryshojmë nga D në E. Nëse E nuk është një solution state, algoritmi do të kthehet prapa dhe do të rizgjidhë decisionin tjetër. Meqenëse fig. 3 është një pemë binare, nuk ka më decision në nyjen C, prandaj algoritmi do të kthehet dhe do të zgjedhë decision nga nyja B, duke bërë algoritmi kalon nga E state në F state. Më pas kjo do të përsëritet derisa e gjithë pema të jetë kërkuar dhe të arrihet një solution state.
Me DFS është e pamundur të dihet nëse zgjidhja që ka gjetur algoritmi është më e mira.

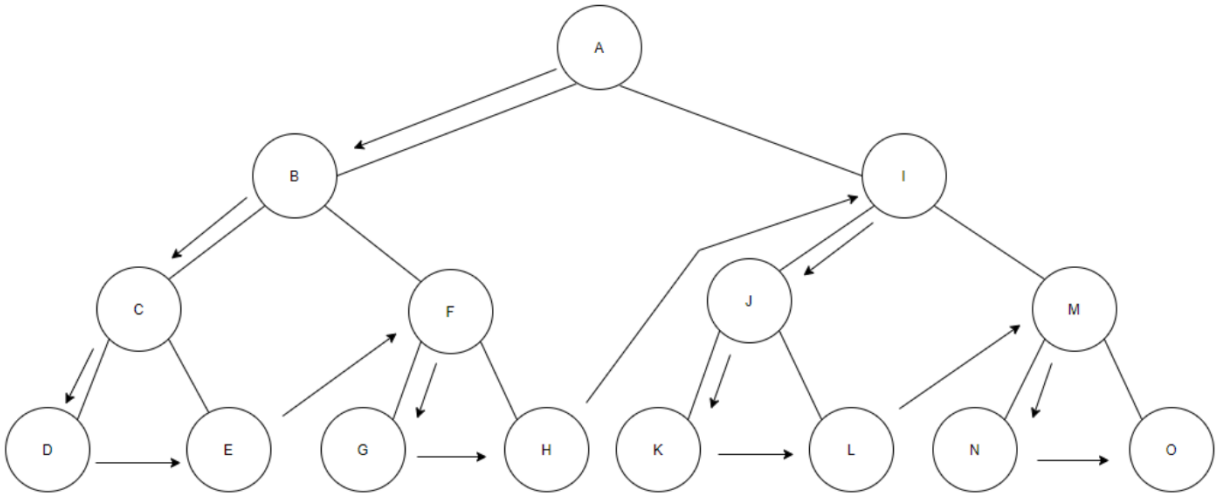


*Figura 3. DFS Alogritmi*

### 3.2.2 Breadth first search

Për të shmangur problemin e mungesës së zgjidhjeve më të mira, mund të përdoret një algoritëm tjetër, siç është BFS. Ky algoritëm do të kërkojë fillimisht nëpër gjerësinë e pemës. Në fig. 4 një shembull se si BFS do të kërkojë përmes një binary state tree, ku A është start state. Në fig. 4 algoritmi zgjedh decisionin e parë, ndryshe nga DFS, BFS do të rizgjidhë më pas nëse zgjedhja e parë nuk ishte solution state, që do të thotë se algoritmi do të kapërcejë nga gjendja B në gjendjen I. Nëse gjendja I nuk është solution state, BFS do të kthehet në decisionin e parë në gjendjen B dhe do të testojë C. Për më tepër nëse C nuk është solution state, algoritmi do të vazhdojë më pas në decisionin tjetër në gjendjen B, duke kaluar në gjendjen F. Më pas kjo do të përsëritet derisa të jetë kërkuar e gjithë pema dhe të gjendet një solution state.

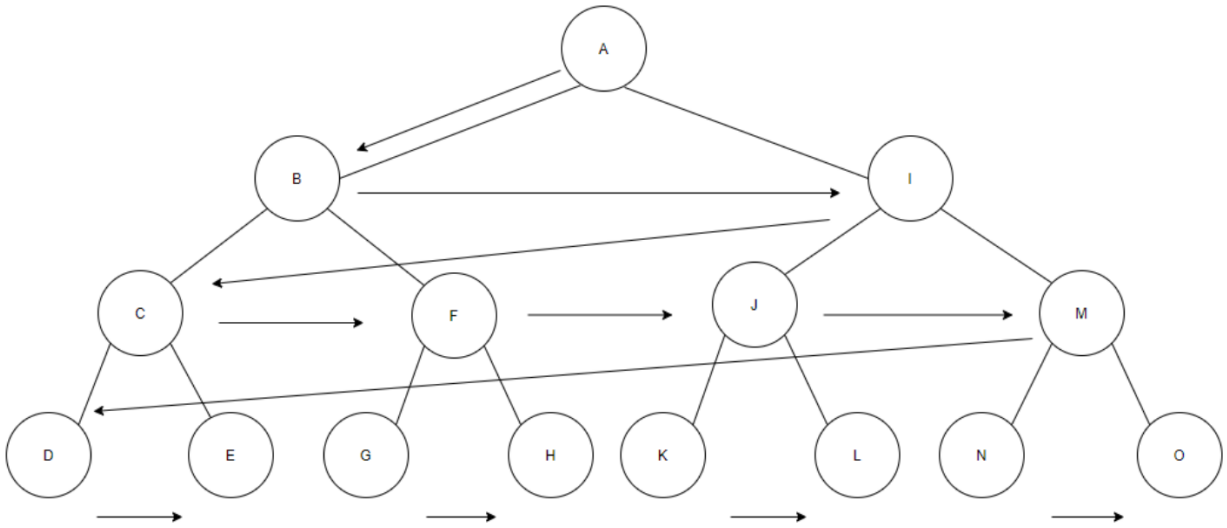


*Figura 4. BFS Alogritmi*

### 3.2.3 Best first search (A*)

Problemi me BFS dhe DFS Algoritmet është se të dy algoritmet mund të përfundojnë duke kërkuar nëpër të gjithë state tree-in nëse state solution gjendet në fund të pemës, e cila në shembujt e paraqitur në fig. 5 dhe 6 do të ishte në O state-in. Andaj përdoren algoritmet me **heuristics**. [2]

Një heuristic vlerëson se sa larg nga goal state është çdo state i caktuar, duke caktuar një vlerë për distancën. Një nga këto algoritme është A*, i cili përdor kërkimin heuristic për të kërkuar se cili decision-i është 'më i miri'. Në A* objektivi kryesor i algoritmit është të gjejë një solution state me vlerën më të ulët heuristic-e të mundshme. Kjo do të thotë që si g ashtu edhe h, në f = g + h, duhet të jenë sa më të ulëta që të jetë e mundur [6].

Në fig. 5 mund të shihet se si A* do të kërkonte në një binary state tree, me **heuristic-en** e secilës nyje të shkruar mbi nyje. Meqenëse A* ka nevojë për një **heuristic-ë** të ulët, në decision e parë mund të zgjedhë midis B, h = 6 dhe J, h = 8, duke e bërë algoritmin të zgjedhë B. Këto dy nyje quhen nyje të hapura. Kur një nyje është zgjedhur, ajo më pas mbyllet [2].

Do të thotë se në decision-in tjetër, algoritmi do të jetë në gjendje të zgjedhë midis: C, h = 5, F, h = 4 dhe J, h = 8. Duke lënë të hapura zgjedhjet e mëparshme për algoritmin, lejohet të zbresë një degë tjetër, nëse dega aktuale rezulton të mos jetë aq e mirë sa u vlerësua fillimisht (kërkon një minimum lokal).
Kjo gjithashtu do të thotë se në rastin tjetër ku algoritmi ka 9 për të zgjedhur një nyje në gjendjen F, algoritmi do të zgjidhte C, pasi zgjedhjet e tij janë: C, h = 5, G, h = 7, H, h = 9 , dhe J, h = 8. Kjo do të vazhdojë më pas derisa algoritmi të gjejë goal state ose të ndalet. Meqenëse efikasiteti i A* varet shumë nga heuristic-a e tij, është e rëndësishme të theksohet se nëse heuristic-a është joefikase, një algoritëm A* gjithashtu mund të përfundojë duke kërkuar nëpër të gjithë pemën. Problemi me A* është se kur ekzekutohet ky algoritëm në state trees më të mëdha, algoritmi përfundon me shumë nyje të hapura, që do të thotë se kërkon shumë memorie për të ruajtur të gjitha zgjedhjet që ka [6].

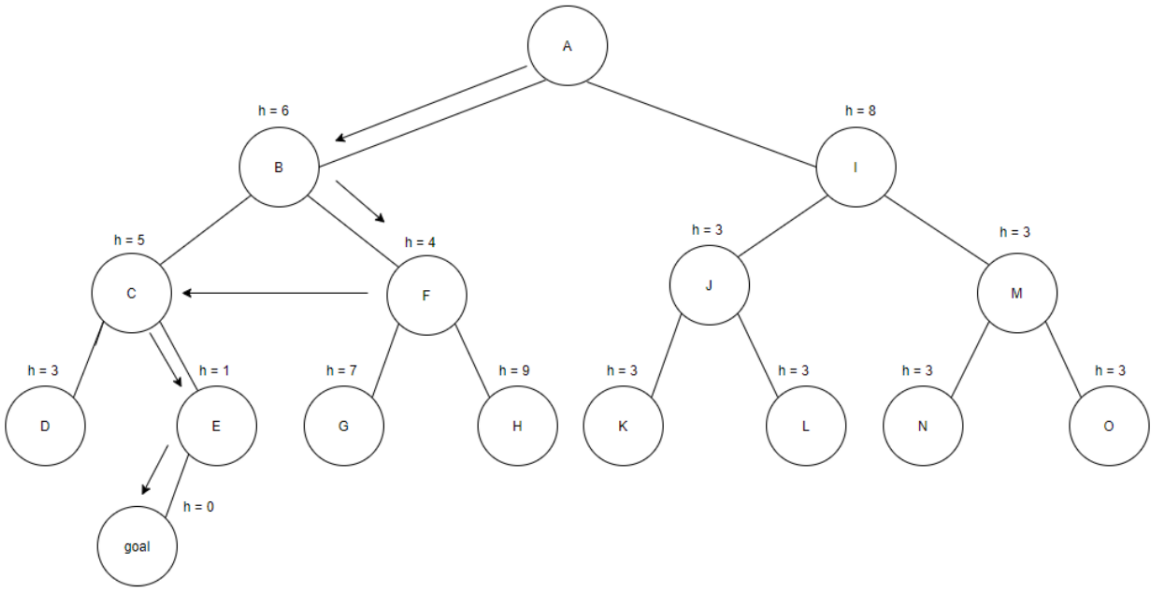


*Figura 5. A* Alogritmi*

### 3.2.4 Iterative Deepening Depth first search (IDDFS)

Për të zgjedhur problemin e memories që u shfaqte tek algoritmi A*, është krijuar algoritmi Iterative Deepening Depth First Search (IDDFS). Duke përdorur iterative deepening, IDDFS arrin një pseudo breadth first.
Në fig. 6, tregohet kërkimi i IDDFS përmes një binary state tree. Këtu ngjyrat e ndryshme tregojnë iterative deepening, ku gjendja e fillimit është A. Fillimi me algoritmin vepron si një DFS normale, megjithëse injoron nyjet që janë më të thella se depth threshold, që është 1 në fillim. Kjo shihet nga shigjetat blu. Kur pema është kërkuar përmes threshold-it të dhënë, threshold-i zgjerohet me një. Kjo

mund të shihet nga shigjetat e verdha. Kur pema kërkohet në threshold-in 2 dhe zgjidhja nuk është gjetur, threshold-i do të rritet përsëri me një. Treguar si shigjeta të kuqe. Duke kërkuar nëpër pemë me IDDFS ju eliminoni problemin e mundshëm të një zgjidhjeje më të mirë duke qenë në një depth më të hershëm, që ka DFS.

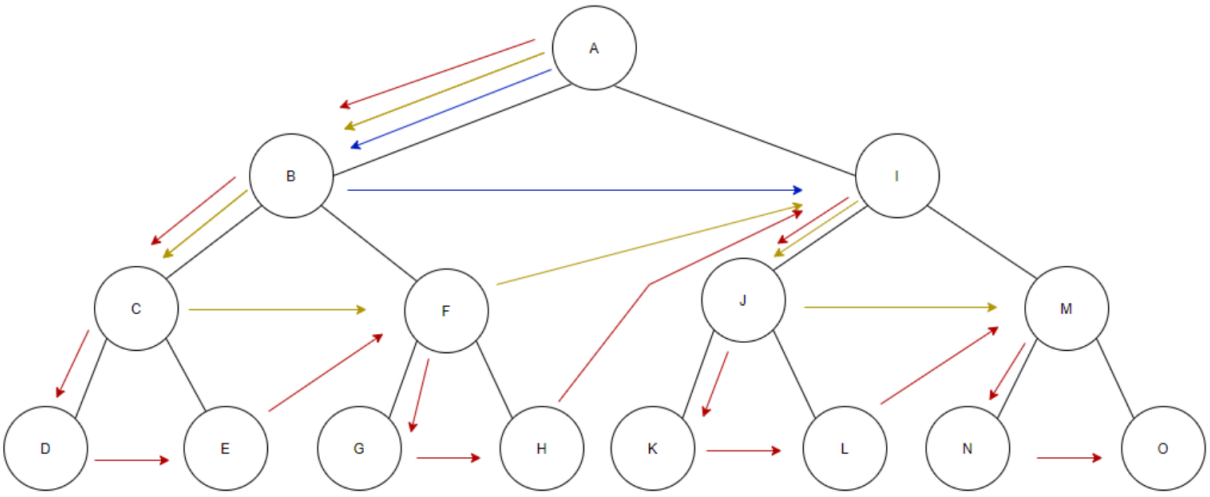


*Figura 6. IDDFS Alogritmi*

### 3.2.5 Iterative Deepening A* (IDA*)

Algoritmi Iterative Deepening A* (IDA*) ndryshe njihet edhe si algoritmi Korf, pasi u zhvillua nga Richard Korf dhe u krijua posaçërisht për të zgjidhur kubin e Rubikut. Ky algoritëm u krijua për tu përsëritur në thellësi, si IDDFS, por edhe për të aplikuar një heuristic si algoritmi A*. Kjo e lejon atë të gjejë gjendjen e zgjidhjes së kubit në kompleksitetin kohor të A*, por të përdorë vetëm kompleksitetin e memories së DFS, pasi të gjitha nyjet e hapura që janë hapur në thellësi të mëparshme janë të mbyllura. Dallimi me IDA* nga një A* normal është se kur arrin në fund të pemës, algoritmi nuk përsëritet në një hap të mëparshëm, threshold-i thjeshtë zgjerohet duke mbyllur gjithashtu të gjitha nyjet e hapura. Për të gjetur një heuristic që mund të gjejë në mënyrë efikase zgjidhjen e një kubi, Korf propozoi që duke përdorur një bazë të dhënash që ka çdo gjendje në të cilën mund të jetë një kub, kjo bazë të dhënash mund të nxjerrë numrin e lëvizjeve në çdo gjendje të caktuar në të cilën ndodhet kubi.

## 3.3 Kompleksiteti kohor

### 3.3.1 Kompleksiteti kohor i rastit më të keq (IDA*)

Rasti më i keq ndodh kur zgjidhja optimale kërkon eksplorimin e një pjese të konsiderueshme të hapësirës së kërkimit të Kubit të Rubikut. Kjo zakonisht ndodh kur konfigurimi fillestar është larg nga gjendja e qëllimit, duke kërkuar një numër të madh lëvizjesh për të arritur zgjidhjen. Kompleksiteti kohor në rastin më të keq është eksponencial, i përshkruar si O(b^d), ku b është faktori i degëzimit(numri i lëvizjeve të disponueshme nga çdo gjendje) dhe d është thellësia e zgjidhjes.

### 3.3.2 Kompleksiteti kohor i rastit mesatar (IDA*)

Rasti mesatar për IDA* në Kubin e Rubikut varet nga faktorë të ndryshëm, duke përfshirë konfigurimin fillestar dhe konfigurimin specifik të qëllimit. Është e vështirë të përcaktohet saktësisht rasti mesatar për shkak të numrit të madh të konfigurimeve të mundshme dhe shpërndarjes jo uniforme të zgjidhjeve brenda hapësirës së kërkimit por, IDA* performon dukshëm më mirë për shkak të eksplorimit të saj të drejtuar nga heuristika.

### 3.3.3 Kompleksiteti kohor i rastit më të mirë (IDA*)

Rasti më i mirë ndodh kur konfigurimi fillestar është tashmë në gjendje të zgjidhur. Në këtë rast, IDA* mund të zbulojë zgjidhjen menjëherë pa kryer asnjë lëvizje ose eksplorim. Kompleksiteti kohor në rastin më të mirë është konstant, i përshkruar si O(1).

### 3.3.4 Kompleksiteti kohor i rastit më të keq (IDDFS)

Kompleksiteti kohor i rastit më të keq për IDDFS në Kubin e Rubikut është i njëjtë me rastin më të keq për IDA*. Mund të arrijë nivele eksponenciale kompleksiteti, të përshkruara si O(b^d), ku b është faktori i degëzimit dhe d është thellësia e zgjidhjes. Meqenëse IDDFS është një kërkim i parë në thellësi me thellim përsëritës, ai eksploron hapësirën e kërkimit në një mënyrë të ngjashme me IDA*, por pa udhëzime heuristike.

### 3.3.5 Kompleksiteti kohor i rastit mesatar (IDDFS)

Kompleksiteti mesatar i kohës së rastit për IDDFS në Kubin e Rubikut është gjithashtu sfidues për t'u përcaktuar me saktësi për shkak të hapësirës së madhe të kërkimit dhe shpërndarjes jo uniforme të zgjidhjeve. Megjithatë, në përgjithësi performon më keq se IDA* sepse eksploron hapësirën e kërkimit pa përfitimin e heuristikës. Rrjedhimisht, kompleksiteti mesatar i kohës së rastit për IDDFS pritet të jetë më i lartë se ai i IDA*. [5]

### 3.3.6 Kompleksiteti kohor i rastit më të mirë (IDDFS)

Kompleksiteti kohor i rastit më të mirë për IDDFS në Kubin e Rubikut, ngjashëm me IDA*, ndodh kur konfigurimi fillestar është tashmë i zgjidhur. Në këtë rast, algoritmi do të përfundonte menjëherë pa kryer asnjë lëvizje, duke rezultuar në një kompleksitet kohor prej O(1).

## 3.4 Kompleksiteti hapësinor

Kompleksiteti hapësinor i IDA* dhe IDDFS në Kubin e Rubikut përcaktohet kryesisht nga thellësia maksimale e pemës së kërkimit dhe strukturat e të dhënave të përdorura për të ruajtur gjendjet dhe për të kryer kërkimin.

### 3.4.1 Kompleksiteti hapësinor i rastit më të keq (IDA*)

Kompleksiteti i hapësirës në rastin më të keq ndodh kur algoritmi duhet të eksplorojë një pjesë të konsiderueshme të hapësirës së kërkimit, duke arritur thellësinë maksimale të pemës së kërkimit. Në këtë rast, kompleksiteti i hapësirës është proporcional me thellësinë maksimale të pemës së kërkimit. Thellësia maksimale përcaktohet nga numri i lëvizjeve të nevojshme për të arritur në gjendjen e qëllimit nga gjendja e fillimit.
Kompleksiteti i hapësirës zakonisht përshkruhet si O(d), ku d është thellësia maksimale e pemës së kërkimit.

### 3.4.2 Kompleksiteti hapësinor i rastit mesatar (IDA*)

Kompleksiteti mesatar i hapësirës së rastit për IDA* në Kubin e Rubikut varet nga faktorë të ndryshëm, duke përfshirë faktorin e degëzimit, shpërndarjen e zgjidhjeve në hapësirën e kërkimit dhe heuristikën e përdorur. Përdorimi i një strategjie përsëritëse të thellimit lejon IDA* të eksplorojë hapësirën e kërkimit në mënyrë graduale, duke reduktuar kërkesat e kujtesës. Kompleksiteti i hapësirës në rastin mesatar mund të përshkruhet si O(d), ku d përfaqëson thellësinë mesatare të arritur gjatë kërkimit.

### 3.4.3 Kompleksiteti hapësinor i rastit më të mirë (IDA*)

Kompleksiteti më i mirë i hapësirës ndodh kur konfigurimi fillestar është tashmë në gjendje të zgjidhur. Në këtë rast, IDA* nuk do të kishte nevojë të kryente ndonjë eksplorim ose të ruante ndonjë gjendje shtesë përtej gjendjes fillestare.
Kompleksiteti i hapësirës në rastin më të mirë është konstant, i përshkruar si O(1).

### 3.4.4 Kompleksiteti hapësinor i rastit më të keq (IDDFS)

Kompleksiteti i hapësirës në rastin më të keq për IDDFS është i ngjashëm me IDA* pasi IDDFS gjithashtu eksploron hapësirën e kërkimit me thellim përsëritës. Kompleksiteti i hapësirës është proporcional me thellësinë maksimale të pemës së kërkimit, e cila përcaktohet nga numri i lëvizjeve të nevojshme për të arritur gjendjen e qëllimit nga gjendja fillestare.
Kompleksiteti i hapësirës në rastin më të keq zakonisht përshkruhet si O(d), ku d është thellësia maksimale e pemës së kërkimit.

### 3.4.5 Kompleksiteti hapësinor i rastit mesatar (IDDFS)

Kompleksiteti mesatar i hapësirës së rastit për IDDFS në Kubin e Rubikut varet nga faktorë të ngjashëm si IDA*, duke përfshirë faktorin e degëzimit dhe shpërndarjen e zgjidhjeve. IDDFS mund të kërkojë më shumë memorie në krahasim me IDA* pasi eksploron hapësirën e kërkimit pa udhëzimin e heuristikës.
Kompleksiteti i hapësirës në rastin mesatar mund të përshkruhet si O(d), ku d përfaqëson thellësinë mesatare të arritur gjatë kërkimit.

### 3.4.6 Kompleksiteti hapësinor i rastit më të mirë (IDDFS)

Kompleksiteti më i mirë i hapësirës për IDDFS ndodh kur konfigurimi fillestar është tashmë i zgjidhur. Në këtë rast, IDDFS do të përfundonte menjëherë pa kryer ndonjë eksplorim ose pa kërkuar memorie shtesë përtej gjendjes fillestare.
Kompleksiteti i hapësirës në rastin më të mirë është konstant, i përshkruar si O(1).

## 3.5 Permbledhje

### 3.5.1 Formulimi i problemit

- **States**
- Per IDFS

```
class State:
    cube = None
    cost = 0
    parent = None
    move = None
```

*Figura 7. IDFS*

- Per IDA*

```
class State:
    cube = None
    g = 0
    h = 0
    parent = None
    move = None
```

*Figura 8. IDA**

- **Actions**

FrontClockwise,FrontAnti-Clockwise, UpClockwise, UpAnti-Clockwise, DownClockwise, DownAnti-Clockwise, LeftClockwise, LeftAnti-Clockwise, RightClockwise, RightAnti-Clockwise, BackClockwise, BackAnti-Clockwise

### 3.5.2 Search algoritmet

- IDFS
- IDA*

### 3.5.3 Heuristics

- h1 <corner_edge_sum_max>
  Admissible
  It returns the Max(Sum_ManhattanDist_of_Corners, Sum_ManhattanDist_of_Edges)
- h2 <corner_db_heuristic>
  Admissible and Consistent
  Storing the corner configurations along with their depths in the database
  Depth is used as heuristic

### 3.5.4 Rezultatet

Nëse përdoren IDFS dhe IDA* me dy heuristics H1 dhe H2 tabela duhet te duket si në vazhdim:

| Depth of goal (d) | Search cost (nodes generated) | | | Effective Branching Factor (b*) | | | Time consumed (seconds) | | |
|---|---|---|---|---|---|---|---|---|---|
| | IDFS | IDA* (h1) | IDA* (h2) | IDFS | IDA* (h1) | IDA* (h2) | IDFS | IDA* (h1) | IDA* (h2) |
| 1 | 12 | 12 | 12 | 12 | 1 | 1 | <1 | <1 | <1 |
| 2 | 107.2941 | 36 | 24 | 11.15385 | 1 | 1 | <1 | <1 | <1 |
| 3 | 1054.526 | 72 | 36 | 10.84579 | 1.5 | 1 | <1 | <1 | <1 |
| 4 | 12270 | 180 | 48 | 10.86 | 1.56 | 1.25 | 2.55 | <1 | <1 |
| 5 | 124585 | 1428 | 588 | 10.8 | 2.27 | 1.6 | 24.8 | <1 | <1 |
| 6 | | 76680 | 1164 | | 3.33 | 1.37 | | 48 | <1 |

*Figura 9. Tabela e rezultateve*

# IV. Konkluzioni

Si përfundim, ky dokumentim ofron një pasqyrë të Kubit të Rubikut, historisë së tij dhe algoritmeve të përdorura për ta zgjidhur atë. Kubi i Rubikut është një enigmë popullore tre-dimensionale që përbëhet nga 27 kube më të vegjël të renditur në një rrjet 3x3x3. Me mbi 43 kuintilion kombinime të mundshme, zgjidhja e Kubit të Rubikut paraqet një sfidë komplekse.

Algoritmet e diskutuara, duke përfshirë "Depth First Search", "Breadth First Search", "Best First Search" (A*), "Iterative Deepening Depth First Search" (IDDFS) dhe "Iterative Deepening A*" (IDA*), janë algoritme kërkimi të përdorura për të gjetur zgjidhje për Kubin e Rubikut. Këto algoritme përdorin heuristika dhe strategji kërkimi për të lundruar në hapësirën e pamasë të kërkimit në mënyrë efikase.

Ndërsa çdo algoritëm ka avantazhet dhe kufizimet e veta, të gjitha ato kontribuojnë në avancimin e fushës së teknikave të zgjidhjes së enigmave dhe zgjidhjes së problemeve.

Po ashtu vlen të ceket se implementimi Python i zgjidhjes së Kubit të Rubikut shfaqi fleksibilitetin dhe aftësinë e gjuhës për zhvillimin e algoritmeve të ndërlikuara dhe manipulimin e strukturave të të dhënave.

# Referencat

[1] Erik D Demaine, Martin L Demaine, Sarah Eisenstat, Anna Lubiw, and Andrew Winslow. Algorithms for solving rubik's cubes. In European Symposium on Algorithms, pages 689–700. Springer, 2011.

[2] Rina Dechter and Judea Pearl. Generalized best-first search strategies and the optimality of a. Journal of the ACM (JACM), 32(3):505–536, 1985.

[3] Richard E. Korf. Finding optimal solutions to rubik's cube using pattern databases. pages 700–705, 1997

[4] Morwen B. Thistlethwaite, 1981. Thistlethwaite's 52-move algorithm. [online] Available at:http://www.jaapsch.net/puzzles/thistle.htm. [Accessed 1 May 2023].

[5] Nail El-Sourani, Sascha Hauke, and Markus Borschbach. An Evolutionary Approach for Solving the Rubik's Cube Incorporating Exact Methods. [PDF] Available at: ¡http://www.genetic-programming.org/hc2010/7-Borschbach/Borschbach-Evo-App-2010-Paper.pdf¿ [Accessed 2 May 2023].