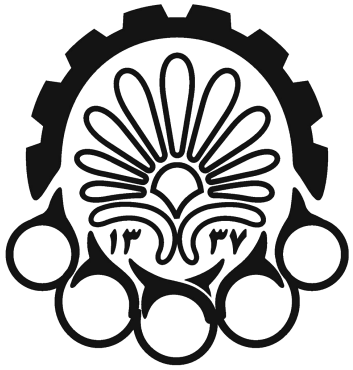


دانشگاه صنعتی امیرکبیر
(پلی تکنیک تهران)

دانشگاه صنعتی امیرکبیر
دانشکده مهندسی کامپیوتر و فناوری اطلاعات

گزارش پروژه اول درس «هوش مصنوعی و سیستم‌های خبره»

حل مسئله با جستجو

امیر حقیقتی ملکی

استاد:
دکتر نیک‌آبادی

بهار ۹۶

چکیده

حل مسئله با جستجوی درخت و در حالت کلی گراف فضای حالت مسئله یکی از روش‌های کلاسیک برای حل مسائلی است که در آن محیط کاملا قابل مشاهده، گسسته، شناخته شده و قطعی می‌باشد. در این روش، عامل‌های مبتنی بر هدف توصیف می‌شوند و با دادن فرموله‌بندی مناسبی از مسئله مورد نظر، می‌توان به راه‌حل (در صورت وجود) دست یافت. طبق تعریف پروژه، فریم‌ورکی[1] برای این منظور پیاده‌سازی شد که در آن از الگوریتم‌های جستجوی گرافی اول سطح، اول عمق (عمق نامحدود، عمق محدود، افزایش تدریجی عمق)، هزینه یکنواخت، دو جهته که از جزو الگوریتم‌های جستجوی ناآگاهانه به حساب آمده و *A که یک الگوریتم جستجوی آگاهانه است، استفاده شد. این فریم‌ورک به زبان Java بوده و مستندات مربوط به استفاده از آن و نحوه فرموله‌بندی مسئله در این گزارش ذکر شده است. سه مسئله نمونه مسیریابی، پازل لغزشی و مبلغان مذهبی و آدم‌خوارها در تعریف پروژه ذکر شده‌اند که با انجام فرموله‌بندی و انجام آزمایشات، پاسخ این مسائل و نتایج عملکرد هرکدام از الگوریتم‌های پیاده‌سازی شده مشخص شدند. الگوریتم جستجوی گرافی اول بهترین حریصانه نیز می‌تواند در آینده به منظور افزودن قابلیت‌های بیشتر به این فریم‌ورک اضافه شود.

کلمات کلیدی: هوش مصنوعی، مسائل کلاسیک، حل مسئله با جستجو، جستجوی آگاهانه، جستجوی ناآگاهانه، جستجوی گراف

[1] framework

فهرست مطالب

۱ مقدمه

عامل‌های هوشمند واکنشی ساده، در محیط‌هایی که تعداد حالات آن زیاد است و در مسائلی که به مسائل جهان واقعی نزدیک است نمی‌توانند بازدهی مناسبی داشته باشند و در نتیجه دیگر سادگی آن‌ها کارآمد نخواهد بود. برای حل مسائل واقعی، می‌توانیم از عامل‌های مبتنی بر هدفی که اعمال و اقدامات خود را پیش از انجام مورد سنجش قرار می‌دهند استفاده کنیم. این عامل‌ها، حالت‌های محیط خود را به دید اتمیک نگاه می‌کنند؛ به این معنی که در هر حالت، حالت فعلی دیگر قابل تقسیم به جزء های کوچک‌تر نیست. در این‌جا به منظور سادگی فرض می‌شود که راه‌حل یک مسئله، همواره دنباله‌ای ثابت از عملیات مختلف است؛ عمل‌های عامل در طول جستجو برای جواب، در اثر ورود ادراکات مختلف عوض نخواهند شد. با این مفروضات، روش‌ها و الگوریتم‌هایی که برای جستجوی راه‌حل در مسائلی از این قبیل در نظر گرفته می‌شوند می‌توانند از نوع آگاهانه یا ناآگاهانه باشند که در ادامه به توضیح آن‌ها پرداخته شده است.

۱-۱ عامل و محیط

عامل‌ها می‌توانند با دریافت ادراکات از محیط، با انجام عمل‌هایی روی محیط تاثیر گذاشته و حالت آن‌را عوض کنند و یا اینکه حالت کنونی خود در مقایسه با محیط را تغییر دهند. طبق تعریف عامل‌های هوشمند، می‌بایست فعالیت‌های این عامل‌ها منجر به بیشینه شدن معیار کارآیی[۲] آن‌ها شود. گاهی اوقات، عامل می‌تواند با داشتن هدفی برای خود و تلاش برای رسیدن به آن هدف، معیار کارآیی خود را آسان‌تر به میزان بیشینه برساند. اهداف می‌توانند با محدود مقاصد عامل از انجام عمل‌های مختلف، در سامان‌دهی رفتار عامل کمک به سزایی داشته باشند. در نتیجه فرموله‌کردن هدف و مقصد مورد نظر – براساس موقعیت و حالت کنونی عامل و معیار کارآیی آن – اولین قدم برای حل مسئله می‌باشد.

هدف برای ما، مجموعه‌ای از چند حالت محیط می‌باشد که عامل می‌تواند با انجام عمل‌های مختلف خود را در آن حالت(ها) قرار دهد و به مقصود خود دست یابد؛ برای این منظور عامل می‌بایست مشخص کند که چه عملیاتی باید انجام دهد.
پیش از تصمیم درباره عمل خاصی، عامل باید بداند که باید چه نوع عمل‌هایی را برای انجام و چه حالت‌هایی از محیط را برای رسیدن باید در نظر بگیرد. فرموله‌کردن مسئله به معنی تعریف و توصیف دقیق حالت‌ها و عملیات ممکن با توجه به حالت‌های هدف در یک محیط است. سطح تجرید عملیات‌ها و همچنین میزان جزئیات توصیف حالت‌های محیط می‌تواند بسته به مسئله متفاوت باشد و باید حتما به این نکته توجه شود که ذکر جزئیات بیش از حد و یا کم‌تر از میزان قابل قبول می‌تواند به عدم موفقیت عامل در دستیابی به پاسخ و رسیدن به هدف شود.

محیط برای تصمیم‌گیری در مورد انجام عملیات مختلف، عامل می‌بایست ابتدا بداند که در حالت کنونی خود مجاز به انجام چه عملیاتی است و نتیجه هر عمل منجر به رسیدن به کدام حالت از محیط می‌شود؛ سپس با بررسی هر عمل و با در نظر داشتن هدف، می‌تواند بهترین تصمیم را بگیرد. بنابراین فرض می‌شود که محیط **کاملا قابل مشاهده** است. همچنین در صورت نامحدود بودن عملیات قابل انجام در هر حالت، امکان تصمیم‌گیری برای عامل در عمل وجود نخواهد داشت. در نتیجه محیط می‌بایست **گسسته** باشد. به این معنی که در هر حالت، تعداد محدودی عمل قابل انجام است. همچنین در فرآیند تصمیم‌گیری، عامل مورد نظر ما می‌بایست قادر به پیش‌بینی نتیجه حاصل از عمل خود باشد. در غیر این‌صورت قادر به تصمیم‌گیری نخواهد بود. به عبارت دیگر فرض می‌شود محیط **شناخته شده** است و عامل می‌داند که با انجام هر عمل از حالت فعلی به کدام حالت می‌تواند برود. همچنین عدم قطعیت اعمال به کلی نادیده گرفته می‌شود و هر عامل در صورت انجام عملی، حتما و الزاما خروجی مربوط به آن عمل را خواهد دید. بنابراین محیط **قطعی** است.

جستجو فرآیند یافتن دنباله‌ای از عملیات که منجر به رسیدن به هدف شود جستجوی راه‌حل نام دارد. یک الگوریتم جستجوی راه‌حل، مسئله‌ای (فضای حالت) را به عنوان ورودی گرفته و راه‌حل آن مسئله را در قالب دنباله‌ای از عملیات و اقدامات توصیف کرده و برمی‌گرداند.الگوریتم‌های مختلفی برای جستجوی راه‌حل در محیط‌های گوناگون وجود دارد که در ادامه به توضیح آن‌ها می‌پردازیم.
هنگامی که راه‌حلی پیدا شد، می‌توان دنباله عملیات ذکر شده در راه‌حل را انجام داد؛ به این مرحله، مرحله اجرا گفته می‌شود. بنابراین برای رسیدن به هدف، می‌بایست ابتدا فرموله‌بندی مناسبی از مسائل و محیط ارائه دهیم، سپس راه‌حل را جستجو کنیم و در مرحله آخر، راه‌حل به دست آمده را برای اجرا به عامل بدهیم.

۱-۲ ساختار مسائل و راه‌حل‌ها

همانطور که در قسمت قبل ذکر شد، به دلیل گرافی بودن فضای حالت مسئله، برای فرموله کردن هر مسئله، گره‌های گراف را در قالب داده‌ساختاری به نام گره ذخیره می‌کنیم که این داده‌ساختار دارای مشخصه‌های زیر است:

[۲] measure Performance

۱. آدرس گره پدر: برای یافتن مسیری که منتهی به این گره شده می‌بایست آدرس گره پدر را در هر گره ذخیره کنیم. این مشخصه در یافتن دنباله حالت‌های جستجو شده در راه‌حل مناسب می‌باشد.

۲. حالت عامل: حالتی که عامل با رسیدن به گره فعلی در آن قرار می‌گیرد در این داده‌ساختار ذخیره می‌شود.

۳. عمل انجام شده: عملی که با انجام آن به گره فعلی رسیده‌ایم و گره فعلی در نتیجه انجام آن عمل در حالت گره پدر می‌باشد. این مشخصه برای به دست آوردن دنباله عملیات انجام شده در طی یک راه‌حل مناسب می‌باشد.

۴. هزینه مسیر تا این گره: هزینه‌ای که تا این رسیدن به این گره پرداخت شده است. این مولفه برای به دست آوردن هزینه مسیر راه‌حل مناسب است.

همچنین به دلیل گرافی بودن فضای حالت مسئله، می‌بایست برای هر مسئله موارد زیر را تعریف کنیم:

۱. ساختار کلی حالت‌ها: اینکه حالت‌ها در قالب ماتریس، عدد، رشته و یا غیره بیان می‌شوند، می‌بایست توسط مسئله بیان شود.

۲. حالت اولیه: حالت اولیه در واقع نشان دهنده پیکربندی[۳] اولیه و حالت شروع عامل می‌باشد. این حالت در جستجوی راه‌حل به عنوان گره آغازین در نظر گرفته می‌شود و در راه‌حل پیدا شده نیز اولین گام خواهد بود.

۳. حالت هدف: درواقع اینکه عامل در مورد مسئله مورد نظر به حالت درستی رسیده یا نه توسط این حالت مشخص می‌شود. مقصد نهایی عامل می‌بایست اینجا باشد. در مسائلی که نتوان حالت هدف را به طور دقیق بیان کرد می‌بایست شرایط حالت هدف ذکر شود.

۴. نتیجه هر عمل در حالت‌های مختلف: از آن‌جا که فرض کرده‌ایم که محیط ما کاملا قابل مشاهده است، مسئله باید گویای نتیجه هر عمل (حالت حاصل شده در ازای انجام عمل) در هر حالتی از فضای حالت مسئله باشد.

۵. عملیات ممکن در هر حالت: پیرو مورد قبلی، از آن‌جا که فرض کرده‌ایم که محیط ما کاملا قابل مشاهده است، مسئله باید گویای عملیات قابل انجام توسط عامل در هر حالتی از فضای حالت باشد.

۶. هزینه انجام هر عمل در هر حالت: عامل باید بداند که هزینه انجام هر عمل در هر حالتی که باشد چقدر خواهد بود.

۷. هزینه مسیری که تا کنون پرداخت شده: عامل باید بداند که تا کنون و تا رسیدن به حالت فعلی چه هزینه‌ای پرداخت کرده است.

۸. تابع شهودی هزینه: این تابع که در واقع تخمینی است از هزینه باقی‌مانده تا رسیدن به حالت هدف، در صورتی باید پیاده‌سازی شود که بخواهیم از روش‌های جستجوهای آگاهانه در یافتن پاسخ استفاده کنیم.

در هر پیمایش و جستجوی راه‌حل، گرهیی ساخته می‌شود که پدر آن گره فعلی بوده و تمامی خصوصیت‌های نام برده شده نیز برای آن ست می‌شوند. اگر در حین جستجوی راه‌حل به گرهی برسیم که رمقصد مورد نظر عامل باشد، برای به دست آوردن دنباله اعمال انجام شده از حالت اولیه، کافی است از گره انتهایی (مقصد) که در جستجوی راه‌حل ساخته شده، شروع کرده و تا زمانی که به گره مبدا نرسیده‌ایم دنباله اعمال و حالت‌ها را بسازیم.

[۳] configuration

۲ نحوه پیاده‌سازی، روش‌ها و الگوریتم‌ها

فریم‌ورک پیاده‌سازی شده توسط نگارنده به زبان Java بوده و GS-ProblemSolver نام دارد که از چهار بسته[۴] به شرح زیر تشکیل شده است:

- بسته `resources`: در این بسته، کلاس‌هایی که به طور مستقیم با پیمایش گراف و ساخته شدن راه‌حل در ارتباط هستند قرار دارند.
- بسته `searchers`: هر جستجوگر که با الگوریتمی منحصر به فرد شروع به جستجو در فضای حالت مسئله می‌کند، در این بسته قرار داده شده است. این بسته حاوی کلاسی تحت عنوان `Searcher` است که به صورت ابسترکت[۵] پیاده‌سازی شده است. برای ثابت نگه داشتن واسط میان جستجوگرها و فضای حالت مسئله و همچنین واسط برنامه‌نویسی کاربری[۶] کلاس‌های جستجوکننده، تمامی جستجوگرها می‌بایست از کلاس `Searcher` به ارث برده شوند[۷].
- بسته `utilities`: شامل کلاس‌های کمکی درکار با فریم‌ورک است. در زمان ویراستاری این گزارش، در این بسته تنها کلاس `GSException` موجود است. این بسته بیشتر برای استفاده در آینده و افزودن کلاس‌های کمکی بیشتر به کار برده می‌شود. به عنوان مثال می‌توان در آینده کلاسی تحت عنوان تحلیل‌گر عملکرد جستجوگرها ساخت و در این بسته قرار داد.
- بسته `main`: که حاوی برنامه اصلی و شروع‌کننده است.

توضیحات کلاس‌های موجود در هر بسته و نحوه پیاده‌سازی آن‌ها در ادامه ذکر می‌شود. شایان ذکر است تمامی توضیحات به صورت کامنت درکد منبع این پروژه نیز قرار دارند و توضیحات ذیل برای شفاف‌سازی بیشتر است.

۲-۱ کلاس‌های اصلی و منبع

همانطور که گفته شد، این کلاس‌ها به‌طور مستقیم با پیمایش فضای حالت مسئله و ساخته‌شدن راه‌حل ارتباط دارند. همگی این کلاس‌ها در بسته `resources` قرار گرفته‌اند و احتمال دارد که با ورود به فصل‌های بعدی کتاب و گسترش یافتن انواع مسئله‌ها و پیچیده‌تر شدن ساختار راه‌حل و نحوه جستجو، ساختار بعضی از کلاس‌ها تغییر یابد.

۲-۱-۱ کلاس مسئله

این کلاس در فایل `Problem.java` به صورت ابسترکت پیاده‌سازی شده و تنها دارای یک متد غیرابسترکت است. توابع این کلاس عبارتند از:

تابع حالت اولیه که در واقع تولیدکننده حالت ابتدایی و آغازین مسئله است. این تابع همچنین به ساختار حالت‌های مسئله قالب نیز می‌دهد و نوع‌داده آن‌ها را مشخص می‌کند.

```
/**
* Getting the problem's initial state in which we start the search.
*
* @return State the initial state.
*/
public abstract State initialState();
```

تابع تست هدف این تابع برای بررسی اینکه آیا حالت داده شده هدف است یا نه مورد استفاده قرار می‌گیرد.

```
/**
* Determine whether the given state is the goal of the problem or not.
*
* @param n State to review.
* @return boolean the answer.
*/
public abstract boolean goalTest(State n);
```

---

[۴] Package
[۵] Abstract
[۶] API
[۷] Inherit

تابع عمل‌های ممکن در هر حالت عملیات ممکن در هر حالت توسط این تابع معلوم شده و برگردانده می‌شود. در این‌جا تمامی اعمال در نتیجه ظاهر نمی‌شوند و فقط اعمالی که قابلیت اجرا در حالت ورودی را دارند در نتیجه برگردانده می‌شوند.

```
/**
* The actions list which can be performed while the agent is in state s.
*
* @param s State to review.
* @return Vector set of actions available in state s.
*/
public abstract Vector<Action> actions(State s);
```

تابع معلوم‌کننده نتیجه هر عمل در یک حالت به‌ازای عمل ورودی و قابل انجام در حالت ورودی، حالت نتیجه شدهدر صورت انجام آن عمل در حالت ورودی برگردانده می‌شود.

```
/**
* Returns the result node of an action performed on agent when it was in state s.
*
* @param s State the state in which the agent is.
* @param a Action the action to perform.
* @return State with parent n.
*/
public abstract State result(State s, Action a);
```

تابع هزینه عمل هزینه انجام یک عمل در حالت ورودی را برمی‌گرداند. فرض براین است که عمل ورودی در حالت ذکر شده قابل انجام است.

```
/**
* Returns the cost of action a in state s.
*
* @param s State in which the action will be performed.
* @param a Action to be performed
* @return double cost of the action.
*/
public abstract double actionCost(State s, Action a);
```

تابع هزینه مسیر هزینه مسیری که از حالت اولیه تا گره فعلی (دربردارنده حالت فعلی) طی شده را برمی‌گرداند.

```
/**
* Returns the path in leaded to node n.
* @param n Node the leaf node.
* @return double path from root to leaf node n.
*/
public abstract double pathCost(Node n);
```

تابع شهودی فقط باید در صورتی پیاده‌سازی شود که از جستجوگرهای آگاهانه می‌خواهیم استفاده کنیم و در واقع این تابع مقدار gScroe را برای هر حالت تخمین می‌زند.

```
/**
* Calculate the heuristic value for state s.
*
* @param s {@link State} the state to evaluate.
* @return int heuristic value.
*/
public abstract double heuristic(State s);
```

تابع بازگرداننده راه‌حل این تابع به صورت غیر ابسترکت پیاده‌سازی شده و به ازای هر ورودی که از جنس یک گره می‌باشد، حالت‌هایی که در هنگام جستجوی راه‌حل به‌دست آمده بودند را برمی‌گرداند. این تابع صرفا یک تابع کمکی است که به دست آوردن راه‌حل یک لایه تجرید بیشتر داشته باشد و آسان‌تر باشد. از ذکر جزئیات پیاده‌سازی این تابع پرهیز می‌شود؛ در صورت داشتن ابهام به کد منبع پروژه رجوع شود.

۲-۱-۲ کلاس گره

این کلاس در فایل `Node.java` پیاده‌سازی شده و وظیفه نگهداری اطلاعات مربوط به مسیری که تا حالت کنونی طی شده است را دارد. شرط مساوی بودن یک گره با گره دیگر نیز مساوی بودن حالت‌های این دو گره است. در برخی جستجوها علاوه بر خصوصیات ذکر شده در قسمت‌های قبل، نیازمند دانستن عمق گره کنونی در گراف نیز هستیم که این خصوصیت از بررسی تعداد گره‌های موجود در مسیر منتهی به این گره به آسانی قابل حصول است و در داخل این کلاس نیز به عنوان یک تابع پیاده‌سازی شده است.

۲-۱-۳ کلاس حالت

که در فایل `State.java` پیاده‌سازی شده است، دارای یک آبجکت[۸] به نام وضعیت[۹] و فقط به خاطر ساختارمند کردن هرچه بیشتر به عنوان یک کلاس پیاده‌سازی شده است. خصیصه وضعیت همان حالت مورد نظر است که به صورت اتمیک است.

۲-۱-۴ کلاس عمل

این کلاس نیز در فایل `Action.java` پیاده‌سازی شده و تنها دارای دو خصیصه `data` و `cost` می‌باشد که اولی توضیحات و داده‌های مربوط به عمل است و دومی هزینه انجام عمل می‌باشد. همانند کلاس حالت، این کلاس نیز صرفا برای ساختارمند کردن بیشتر پروژه، پیاده‌سازی شده است.

۲-۲ جستجوگرها

این کلاس‌ها در بسته `searchers` قرار داشته و وظیفه یافتن راه‌حل برای مسئله پاس داده شده به آن‌ها در هنگام ساخته شدن را دارند.

۲-۲-۱ جستجوگر ابسترکت

همانطور که ذکر شد، تمامی جستجوگرها می‌بایست از کلاس ابسترکتی به نام Searcher به ارث برده شوند. این کلاس در فایل `Searcher.java` قرار گرفته است و دلیل اهمیت آن نیز دارا بودن اینترفیس و واسط بسیار ساده و همچنین خصیصه‌هایی که مورد نیاز تمامی جستجوگرهاست، می‌باشد. این کلاس دارای یک متد ابسترکت است که وظیفه آن جستجوی راه‌حل در مسئله داده شده است. هر جستجوگر می‌بایست جداگانه و با توجه به الگوریتم مورد نظر خود، این متد را پیاده‌سازی کند.

```
/**
* The actuator function and main responsibility of this class.
*
* @return Null if no solution found | {@link Node} containing a solution.
* @throws GSException if the frontier is not initialized.
*/
public abstract Node search() throws GSException, InterruptedException;
```

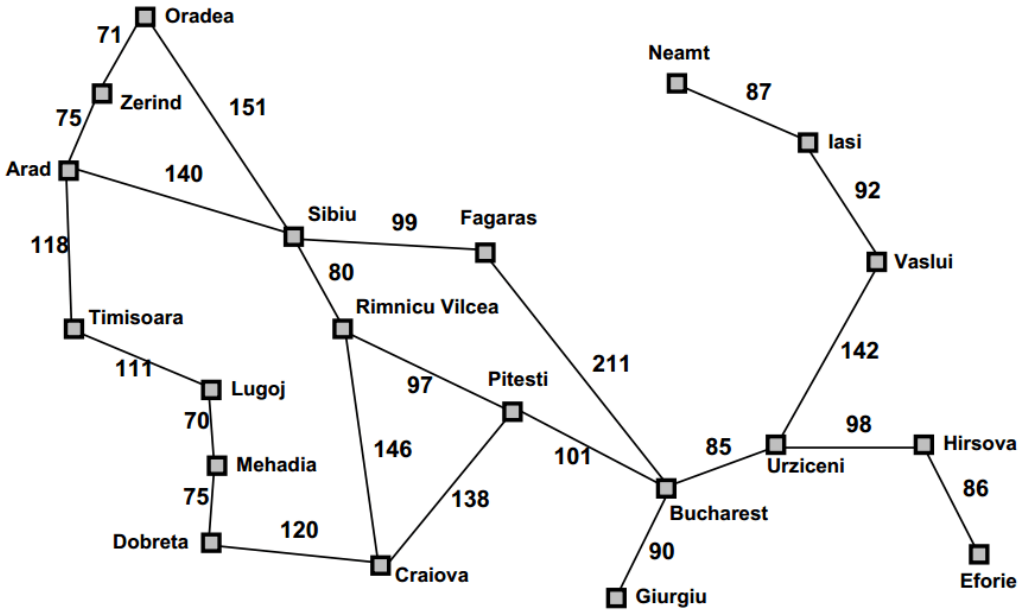
برای ایجاد هر نمونه جستجوگر می‌بایست مسئله را در تابع سازنده به جستجوگر پاس دهیم و سپس متد `search()` را فراخوانی کنیم.
به علاوه، این کلاس دارای متغیرهای دیگری نیز هست که برای شمارش تعداد گره‌های مشاهده شده، تعدا گره‌های بسط‌داده شده و حداکثر حافظه مورد استفاده در طول جستجو به کار برده می‌شوند.

---
[۸] Object
[۹] Status

۳ آزمایشات و نتایج

سه مسئله نمونه طرح شده در تعریف پروژه به شرح گفته شده فرموله‌بندی شدند که جزئیات فرموله‌بندی آن‌ها در فایل `Main.java` قابل مشاهده است. با به راه‌اندازی هر جستجوگر مورد نظر در صورت پروژه در مورد هر مسئله نتایجی به دست آمد که در ادامه شرح داده می‌شوند.

۳-۱ مسئله اول: مسیریابی شهرهای رومانی



شکل ۱: نقشه شهرهای رومانی

جزئیات فرموله‌کردن و مفروضات مورد استفاده در فرموله کردن این مسئله در بخش پیوست آ-۱ قابل مشاهده است.

۳-۱-۱ جستجوی سطح اول

آزمایش جستجوی سطح اول روی این مسئله با فرموله‌بندی موجود در پیوست آ-۱ انجام شد و نتایج این آزمایش در جدول ۱ قابل مشاهده است.

جدول ۱: نتایج جستجوی گراف شهرهای رومانی با استفاده از الگوریتم اول سطح.

معیار	نتیجه به دست آمده طی اجرای اگوریتم
بهترین مسیر یافته شده	Arad, Sibiu, Fagaras, Bucharest, Urziceni, Vaslui
هزینه مسیر یافته شده	۶۷۷ واحد
تعداد گره‌های بسط داده شده در حین جستجو	۸۴
تعداد گره‌های مشاهده شده در حین جستجو	۳۳
حداکثر حافظه استفاده شده در حین جستجو	۴۵

۳-۱-۲ جستجوی عمق اول با عمق محدود ۸

در پیاده‌سازی جستجوگر عمق اول در این فریم‌ورک، می‌بایست قبل از شروع جستجو مشخص شود که آیا جستجوی مورد نظر دارای عمق محدود است یا خیر. با ست کردن عمق ۸ برای این جستجوگر، نتایج آزمایش در جدول ۲ قابل مشاهده است.

جدول ۲: نتایج جستجوی گراف شهرهای رومانی با استفاده از الگوریتم اول عمق با عمق محدود ۸.

معیار	نتیجه به دست آمده طی اجرای اگوریتم
بهترین مسیر یافته شده	Arad, Zerind, Oradea, Sibiu, Fagaras, Bucharest, Urziceni, Vaslui
هزینه مسیر یافته شده	۸۳۴ واحد
تعداد گره‌های بسط داده شده در حین جستجو	۲۲
تعداد گره‌های مشاهده شده در حین جستجو	۱۴
حداکثر حافظه استفاده شده در حین جستجو	۱۴

۳-۱-۳ جستجوی *A

در این جستجو، تابع شهودی مورد استفاده، فاصله واقعی هر شهر تا شهر مقصد بوده که از وب نتایج یافت شدند. جزئیات این تابع در پیوست آ-۱ قابل مشاهده است. نتایج این آزمایش نیز در جدول ۳ ذکر شده است.

جدول ۳: نتایج جستجوی گراف شهرهای رومانی با استفاده از الگوریتم اول عمق با عمق محدود ۸.

معیار	نتیجه به دست آمده طی اجرای اگوریتم
بهترین مسیر یافته شده	Arad, Sibiu, Fagaras, Bucharest, Urziceni, Vaslui
هزینه مسیر یافته شده	۶۷۷ واحد
تعداد گره‌های بسط داده شده در حین جستجو	۳۲
تعداد گره‌های مشاهده شده در حین جستجو	۱۲
حداکثر حافظه استفاده شده در حین جستجو	۲۴

۳-۲ مسئله دوم: پازل ۸تایی

۳-۲-۱ جستجوی اول عمق گرافی

در جستجوی اول عمق گرافی نمونه داده شده در تعریف پروژه به مشکل عدم وجود حافظه کافی برخورد کردیم که با تغییر نمونه ورودی به ورودی زیر (فقط به عنوان تست) می‌توان پاسخ مناسب از آن گرفت: $1, 0, 2, 3, 4, 5, 6, 7, 8$

۳-۲-۲ جستجوی دو جهته

با اجرای این الگوریتم با حالت اولیه ذکر شده در تعریف پروژه، که زمان بسیار زیادی برای اجرا نیاز داشت، به نتیجه‌ای نرسیدیم. اما با دادن حالت $1, 4, 2, 0, 3, 5, 6, 7, 8$ به عنوان حالت اولیه، به نتایج ذکر شده در جدول ۴ رسیدیم.

جدول ۴: نتایج جستجوی دو جهته فضای حالت پازل ۸ تایی.

معیار	نتیجه به دست آمده طی اجرای اگوریتم
بهترین مسیر یافته شده	012345678, 102345678, 142305678, 142035678
هزینه مسیر یافته شده	۳
تعداد گره‌های بسط داده شده در حین جستجو	۹
تعداد گره‌های مشاهده شده در حین جستجو	۵
حداکثر حافظه استفاده شده در حین جستجو	۱۱

۳-۲-۳ جستجوی *A با تابع شهودی فاصله مستقیم

با اجرای الگوریتم داده شده و تابع شهودی مورد نظر، با حالت اولیه ذکر شده در قسمت قبل، در تعریف پروژه به نتایج ذکر شده در جدول ۵ رسیدیم.

جدول ۵: نتایج جستجوی دو جهته فضای حالت پازل ۸ تایی.

معیار	نتیجه به دست آمده طی اجرای اگوریتم
بهترین مسیر یافته شده	012345678, 102345678, 142305678, 142035678
هزینه مسیر یافته شده	۳
تعداد گره‌های بسط داده شده در حین جستجو	۱۲
تعداد گره‌های مشاهده شده در حین جستجو	۵
حداکثر حافظه استفاده شده در حین جستجو	۱۲

۳-۳ مسئله سوم: مبلغین مذهبی و آدم‌خوارها

با پیاده‌سازی و فرموله‌بندی این سوال در متد p3 فایل `Main.java` به اجرای الگوریتم‌های جستجوی مختلف روی این سوال پرداخته شد.

۳-۳-۱ جستجوی سطح اول

نتایج جستجوی سطح اول در جدول ۶ ذکر شده است.

جدول ۶: نتایج جستجوی سطح اول مسئله مبلغیم مذهبی و آدم‌خوارها

معیار	نتیجه به دست آمده طی اجرای اگوریتم
بهترین مسیر یافته شده	(در سمت اولیه رود) ۳ آدم‌خوار و ۳ مبلغ، ۱ آدم‌خوار و ۳ مبلغ، ۱ آدم‌خوار و ۱ مبلغ، ۰ آدم‌خوار و ۰ مبلغ
هزینه مسیر یافته شده	۳
تعداد گره‌های بسط داده شده در حین جستجو	۳۲
تعداد گره‌های مشاهده شده در حین جستجو	۲۰
حداکثر حافظه استفاده شده در حین جستجو	۳۳

۳-۳-۲ جستجوی عمق اول با افزایش تدریجی عمق

نتایج جستجوی عمق اول با افزایش تدریجی عمق در جدول ۷ قابل مشاهده است.

جدول ۷: نتایج جستجوی سطح اول مسئله مبلغیم مذهبی و آدم‌خوارها

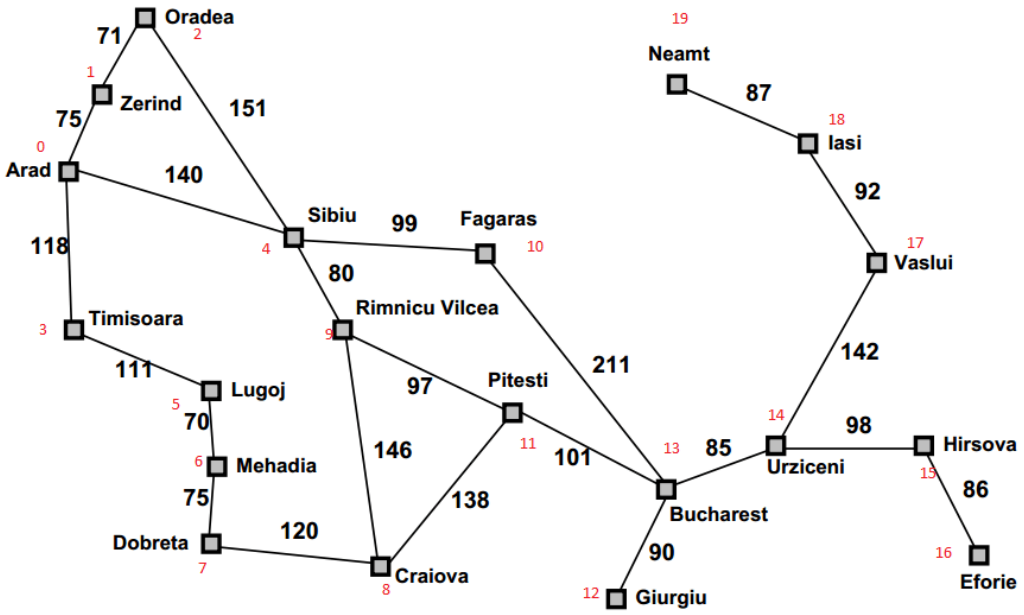
معیار	نتیجه به دست آمده طی اجرای اگوریتم
بهترین مسیر یافته شده	(در سمت اولیه رود) ۳ آدم‌خوار و ۳ مبلغ، ۱ آدم‌خوار و ۳ مبلغ، ۱ آدم‌خوار و ۱ مبلغ، ۰ آدم‌خوار و ۰ مبلغ
هزینه مسیر یافته شده	۳
تعداد گره‌های بسط داده شده در حین جستجو	۵۸
تعداد گره‌های مشاهده شده در حین جستجو	۶۱
حداکثر حافظه استفاده شده در حین جستجو	۲۸

آ پیوست

آ-۱ مسئله اول: مسیریابی شهرهای رومانی

هرکدام از شهرها به ترتیب ذکر شده در شکل ۲ شماره‌گذاری شدند. در تابع شهودی مورد استفاده، فاصله واقعی هرکدام از شهرها نسبت به مقصد داده شده مورد استفاده قرار می‌گرفت که به ترتیب ذکر شده عبارتند از (عدد نوشته شده در هر سطر بیانگر فاصله واقعی آن شهر –مقدار تابع شهودی– تا مقصد است.):

```
710,
634,
576,
677,
408,
636,
720,
669,
561,
507,
330,
447,
388,
324,
267,
283,
387,
0,
66.6,
133,
```



شکل ۲: شماره‌گذاری شهرهای رومانی.