دانشگاه اصفهان

دانشکده مهندسی کامپیوتر

عنوان

گزارش پروژه پایانی

هوش مصنوعی و سیستم‌های خبره

استاد درس

دکتر حسین کارشناس

گردآورنده

علیرضاپرچمی

گرایش

فناوری اطلاعات

نیمسال اول ۱۳۹۷

فهرست مطالب

* آدرس صفحه GitHub جهت مشاهده تمامی Commitها:

https://github.com/AlirezaParchami/Hybrid-Genetic-Algoithm

مقدمه

این پروژه شامل سه بخش است که عبارت است از:

۱- الگوریتم ژنتیک و بهبود هر ژن با استفاده از جستجوی تپه‌نوردی (الگوریتم ژنتیک ترکیبی)

۲- الگوریتم جستجوی موضعی *A برای یافتن بهترین جواب ممکن

۳- مساله بهینه‌سازی (ارضا) قید

در این گزارش، مراحل کار هر قسمت از پروژه به صورت کلی توضیح داده شده است و برخی از نتایج بدست آمده نیز آورده شده است.

به دلیل تعریف خاص مساله و حالت خاص آن برای نمایش افراد یک جمعیت، تصمیم بر آن شد که تمامی قسمت‌های این پروژه را به صورت شخصی برنامه کرده و تنها برای الگوریتم جستجوی *A از یک منبع کمک گرفته شده است. لازم به ذکر است[1] که پروژه موجود در این منبع برای بازی pacman بوده و تمامی بخش های مهم آن از جمله تابع goal_test، تخمین هزینه باقیمانده و نحوه نمایش افراد در این مساله توسط بنده تغییر کرده و متناسب با مساله knapsack بازتعریف شده.

الگوریتم ژنتیک ترکیبی

روند کلی

در ابتدا دو فایل لازم به جهت تعیین پارامترها و مشخص کردن وزن و ارزش اشیا کوله‌پشتی را خوانده و سپس اقدام به تولید یک جمعیت تصادفی میکنیم. تابع population_generation و individual_generation این وظیفه به عهده دارند. تابع population_generation به تعداد popSize تابع individual_generation را فراخوانی می‌کند. تابع individual_generation نیز رشته‌های تصادفی با طول indivSize برمی‌گرداند.

مقدار برازندگی این نسل محاسبه و در متغیر person_fitness نگهداری می‌شوند. سپس این افراد را به تابع reproduction برای تولید فرزندان میدهیم. تابع parent_selection برحسب متغیرهایی که کاربر در ابتدا مشخص کرده است (parentPercent, offsprongPercent)، لیستی از زوج‌ها را برمیگرداند تا عملیات crossover بروی آنها انجام شده و سپس بروی هر رشته بدست آمده mutation اعمال شود.

---

[1] https://gist.github.com/jamiees2/5531924#file-astar-py-L24

پس از اعمال crossover و mutation برحسب احتمال های داده شده (crossoverProb,mutationProb)، فرزندان را برای بهبود توسط الگوریتم تپه‌نوردی به تابع hill_climbing میفرستیم. روند این تابع به صورت بازگشتی کار می‌کند به این صورت که در ابتدا تمامی همسایه‌های یک فرد پیدا شده و برازندگی آن‌ها محاسبه می‌شود. سه حالت امکان دارد رخ بدهد:

۱- چنانچه تمامی همسایه‌ها برازندگی کمتری از رشته فعلی داشتند، رشته فعلی برگردانده می‌شود.

۲- چنانچه ارزش همسایه‌ای برابر باشد، رشته فعلی را وارد لیست tabu می‌کند و سپس مقدار sideway را یک واحد افزایش داده و سپس دوباره تابع hill_climbing اینبار با رشته همسایه و با لیست tabu موردنظر فراخوانی می‌شود.

۳- چنانچه ارزش همسایه‌ای بیشتر از ارزش رشته فعلی باشه، لیست tabu پاک شده، متغیر sideway ریست شده و تابع hill_climbing با رشته جدید، tabu و متغیر sideway ریست شده فراخوانی می‌شود.

این روند تا جایی ادامه می‌یابد که مقدار مشخص شده برای بهبود رشته برسیم و یا رشته با ارزش بیشتری نباشد و یا محدودیت sideway محقق شود.

پس از این قسمت رشته‌های بهبود داده شده به دست می‌آیند و به جمعیت اولیه ما اضافه می‌شود. حال جهت انتخاب popSize عدد رشته با بیشترین برازندگی، تابع replacement را فراخوانی میکنیم که لیست مرتب شده افراد براساس برازندگی آن‌ها را می‌دهد.

سپس مقدار برازندگی بهترین شخص، میانگین برازندگی نسل آخر و تعداد فراخوانی تابع برازندگی در result نگهداری می‌شود تا در انتها برنامه در فایل نمایش داده شوند.

نتیجه

نتایج بدست آمده در فایلی در همان مسیر به اسم log*datetime*.txt ذخیره می‌شوند که حاوی مقدار بهترین شخص، میانگین برازندگی نسل آخر و تعداد فراخوانی تابع برازندگی در هر اجرا هستند و در پایان مقدار میانگین برازندگی بهترین شخص در کل اجراها، میانگین متوسط برازندگی نسل آخر و میانگین تعداد دفعات فراخوانی تابع برازندگی هستند. شکل زیر نمونه‌ای از این فایل است.

```
/////RUN:0
Best Fitness: 1024
Average Fitness: 1024.0
Fitness Callback: 21867
/////RUN:1
Best Fitness: 1024
Average Fitness: 1024.0
Fitness Callback: 18467
/////RUN:2
Best Fitness: 1024
Average Fitness: 1024.0
Fitness Callback: 26848
/////RUN:3
Best Fitness: 1024
Average Fitness: 1024.0
Fitness Callback: 20693
/////RUN:4
Best Fitness: 1024
Average Fitness: 1024.0
Fitness Callback: 26153
==========Final Results:
Average Fitness: 1024.0
Average Average Fitness: 1024.0
Average Fitness Callback: 22805.6
```

الگوریتم A*

روند کلی

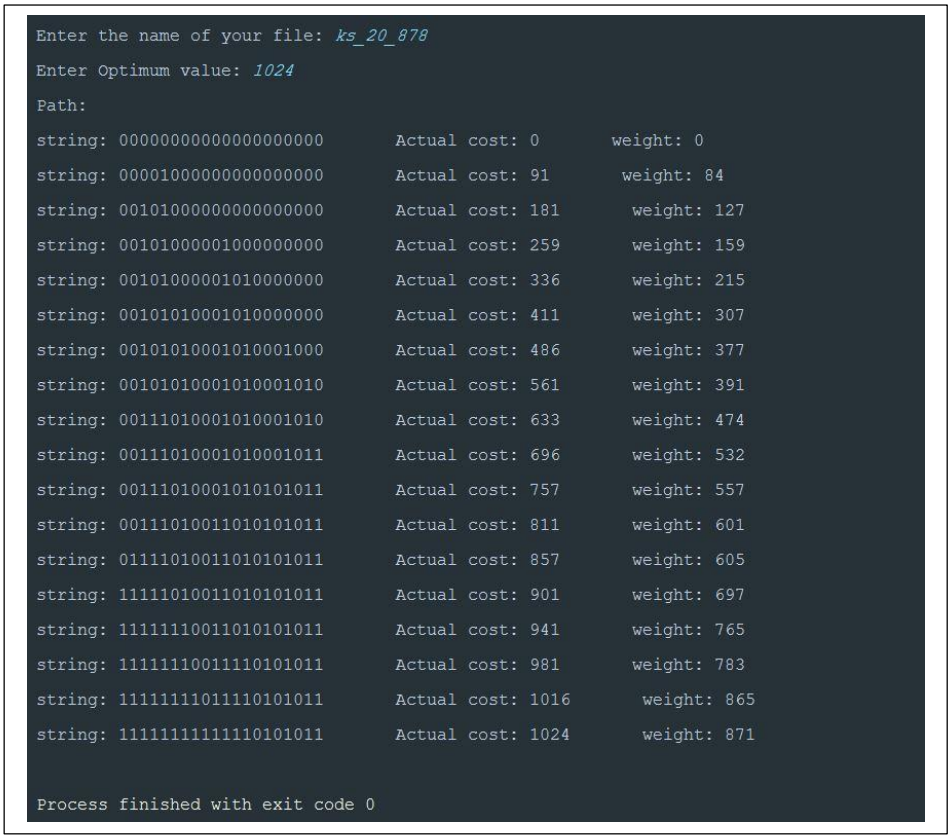
در این الگوریتم یک کلاس به نام Node داریم که حاوی رشته بیتی (افراد)، گره پدر، مقدار هزینه تخمین زده‌شده، مقدار هزینه پیموده شده و وزن رشته است. در این پیاده‌سازی، ساختار داده‌ها همانند درخت است زیرا در عین حال که برخی شاخه‌ها تا عمق‌های بسیار زیادی پایین رفته، ممکن است شاخه‌های دیگر در عمق‌های متفاوتی باشند و ما باید در هر لحظه به همه این گره‌ها دسترسی داشته باشیم تا بتوانیم طبق الگوریتم A*، بهترین گره (بیشینه مقدار تابع هیوریستیک) را انتخاب کنیم.

دو مجموعه (set) در این الگوریتم درنظر گرفتیم که openset به معنا گره‌هایی است که باید آنها را بررسی کنیم و جواب موردنظر ممکن است در هریک از این گره‌ها باشد و closeset به معنا گره‌هایی است که بررسی شده‌اند و مطمئن شده‌ایم که جواب در این شاخه یا فرزندان آن نیستند.

در تابع Goal_test نیز همواره اگر ارزش رشته موردنظر بیشتر یا مساوی مقدار بهینه داده شده بود و وزن آن کمتر یا مساوی maxWeight بود، مقدار True برگردانده و رشته را هدف می‌دانیم.

نتیجه

خروجی این برنامه به صورت درختی می‌باشد. به این صورت که به ترتیب از رشته 000..000 که رشته اولیه ما بوده نمایش داده می‌شود و تمامی رشته‌هایی که در مسیر انتخاب شده‌اند را به همراه مقدار هزینه واقعی و وزن آنها نمایش میدهد. طبیعتا آخرین رشته نشان داده شده، جواب موردنظر ما است. نمونه‌ای از خروجی به شکل زیر است:



مقایسه تعداد فراخوانی تابع knapsack در مقایسه با الگوریتم ژنتیک ترکیبی به صورت زیر است:

الگوریتم A*	الگوریتم ژنتیک
250	18000

البته این مقدار برای فراخوانی تابع knapsack در الگوریتم ژنتیک به دلیل عدم ذخیره و فراخوانی بیش از حد در برخی از شرایط است. ولی در کل فراخوانی knapsack در الگوریتم A* بسیار پایین‌تر از الگوریتم ژنتیک ترکیبی است. یکی از دلایل فراخوانی زیاد در الگوریتم ژنتیک، به دلیل mutation و

crossover است که ما را مجبور به فراخوانی تابع knapsack میکند. همچنین بالابودن مقدار maxGen نیز بسیار تاثیرگذار است چنان‌که اگر maxGen حدود ۱۰یا ۱۵ باشد، تعداد فراخوانی به حدود 8000 بار کاهش می‌یابد.

ذکر این نکته بسیار مهم است که در الگوریتم ژنتیک، به دلیل انتخاب با توزیع یکنواخت، برای دیتاست دوم و سوم نتیجه‌ای نمیدهد در صورتیکه الگوریتم *A بروی دیتاست دوم جواب میدهد. اگرچه مدت زمان برای یافتن جواب کمی طولانی است.

مساله بهینه‌سازی قید (بخش تکمیلی)

روند کلی

در این مساله که از پیچیدگی و حالت‌های متفاوتی شامل می‌شد، نحوه نمایش را به صورت بیت فرض کردم به این صورت که پس از خواندن فایل، تعداد استادها، تعداد درس‌ها و... را به صورت دقیق مشخص کرده و با استفاده از لگاریتم گرفتن تعداد آنها، مشخص میشود که چند بیت برای نمایش استاد، درس و... داریم. برای مثال اگر ۱۲ درس تعریف شده باشد، ۴ بیت برای نمایش هر درس نیاز داریم.

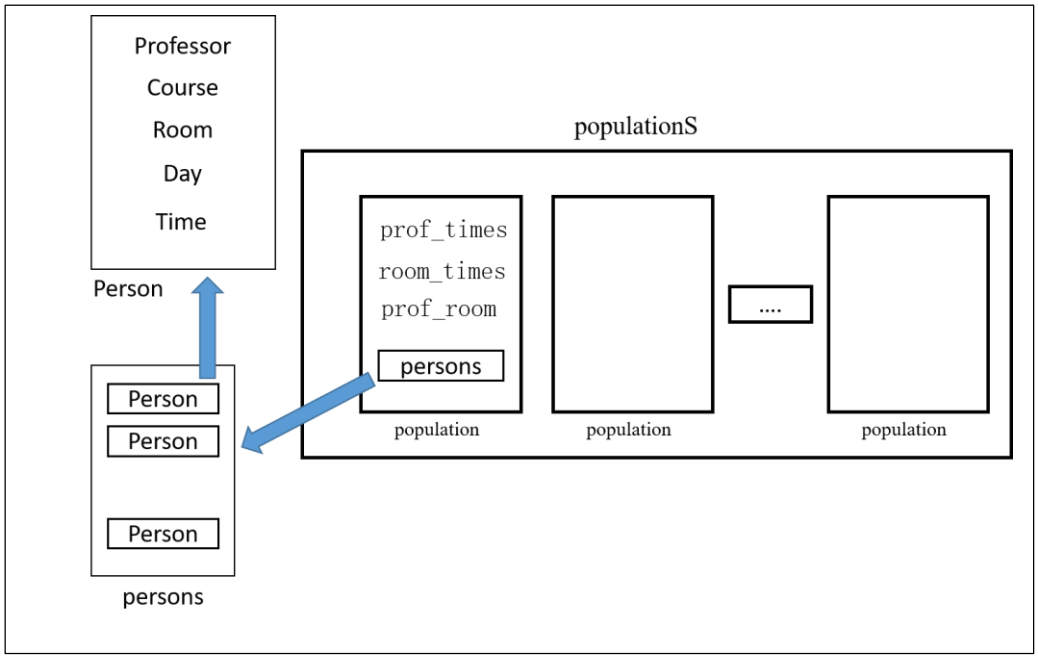
ضمن اینکه قبل از به‌دست آوردن مقدار بیت‌های موردنیاز برای درس، بررسی میکنیم که آیا تعداد دانشجویان ثبت نامی در یک درس از حجم بزرگترین کلاس بیشتر است یا خیر ؟ اگر تعداد دانشجویان ثبت نامی برای یک درس بزرگتر از بیشترین ظرفیت کلاس‌ها بود، آن درس را به course هایی با همان نام اما ظرفیت کمتر تبدیل میکنم برای مثال اگر برای درس هوش مصنوعی ۱۱۰ دانشجو ثبت نام کرده و بیشترین ظرفیت در کلاس‌ها ۴۰ است، کلاس درس هوش مصنوعی به سه کلاس با ظرفیت‌های ۴۰،۴۰ و ۳۰ شکسته می‌شود. به این صورت زمان‌بندی دقیق‌تر و واقعی‌تری برای کلاس‌ها خواهیم داشت. ضمن اینکه اگر چندین استاد قابلیت تدریس درس هوش مصنوعی را داشته باشند، این امکان وجود دارد که سه کلاس بین آنها تقسیم شود تا قیدهای ترجیحی مساله برآورده شوند. (به دلیل کاهش هزینه، اساتید تعداد روز کمتری کلاس برگزار کنند.)

ساختار استفاده شده به این شکل است که هر Person بیانگر یک درس با استاد مشخص در یک کلاس، روز و زمان مشخص است بنابراین کلاس Person شامل متغیرهای Professor، Course، Room، Day، Time است.

هر Population بیانگر یک زمان‌بندی کامل است و می‌تواند یک جواب مساله باشد. کلاس Population دارای یک لیست از Person به نام persons است. هم‌چنین برای اطلاعات موردنیاز که "هر استاد در هر اتاق چندبار تدریس داشته"، "هر استاد در هر روز در چه بازه‌های زمانی کلاس دارد" و "هر اتاق در هر روز در چه زمان‌هایی اشغال هستند" نیاز به سه آرایه چند بُعدی prof_room، prof_times و room_times داریم. به دلیل اینکه در ادامه الگوریتم بروی این رشته‌ها crossover و mutation انجام می‌دهیم، مقادیر ارایه‌های prof_room، prof_times و room_times باید آپدیت شوند که سه تابع موجود در کلاس Population همین وظیفه را بر عهده دارند.

```
Class Population:{
Array prof_times [Profs_size, Days_size, Span_size]
Array room_times [Rooms_size, Days_size, Span_size]
Array prof_room [Profs_size, Rooms_size]
Function update_prof_times()
Function update_room_times()
Function update_prof_room()
persons = []
fitness = int
}
```

همچنین جواب‌های مختلف مساله (زمان‌بندی‌ها) درون متغیر populationS ذخیره می‌شوند. با توجه به مطالب گفته شده، ساختار زیر در مساله استفاده می‌شود.



در تابع generate_persons، برای هر درس یک استاد، یک کلاس، یک روز و یک زمان مشخص می‌کنیم. نکته قابل توجه اینکه ابتدا برای هر درس توسط تابع ()available_profs، استادهایی که توانایی تدریس آن درس را دارند را پیدا میکنیم. سپس از بین این اساتید، به صورت رندوم یکی از آنها را انتخاب میکنیم.

پس از آن، با توجه به تعداد دانشجویان ثبت نامی برای درس، کلاس‌های مناسب را پیدا می‌کنیم. در این قسمت، تابع هیوریستیکی تعریف کرده‌ام که سعی در برآورده کرده هرچه بهتر قیدهای ترجیحی دارد و از بین کلاس‌های ممکن، کلاسی را انتخاب میکند که آن استاد در آن کلاس بیشتر حضور داشته است (به دلیل محدودیت هزینه موسسه). برای انتخاب روز درس نیز به همین صورت عمل کرده و روز هایی را انتخاب می‌کنیم تا استاد روزهای کمتری مجبور به تدریس باشد.

مراحل مربوط به crossover و mutation در تابع reproduction_cop انجام می‌شود و فرزندان جدید تولید شده را برمیگرداند. در مراحل mutation و crossover، پس از تغییر، تمامی شرایط ممکن را در تابع ()check_condition بررسی می‌کنم. این شرایط عبارت است از:

"استاد، درس، کلاس و روز تولید شده نامعتبر(خارج از محدوده تعریف شده) نباشد"

"آیا این استاد میتواند این درس را تدریس کند؟"

"آیا این استاد در این روز و ساعت وقت آزاد دارد ؟"

"آیا این درس در این کلاس میتواند برگزار شود ؟"

نتیجه

همچنین با توجه به شکل نمایش به صورت رشته‌ای، اگرچه نمی‌توان با تعداد جمعیت کم برای یک دیتاست بزرگ برنامه‌ریزی جامع و کاملی ارائه داد اما هیچ‌یک از قیدها نقض نخواهند شد. ضمن آنکه با توجه به تابع‌های هیوریستیک تعریف شده برای انتخاب روز برگزاری درس و کلاس موردنظر، این قیدها رعایت خواهند شد.

فایل موجود در پوشه python با نام COP_log نشان‌دهنده تمامی موارد خواسته شده از جمله بهترین زمان‌بندی در هر اجرا، مقدار تابع برازندگی در هر اجرا، بهترین زمان‌بندی در بین تمامی اجراها و... می‌باشد.