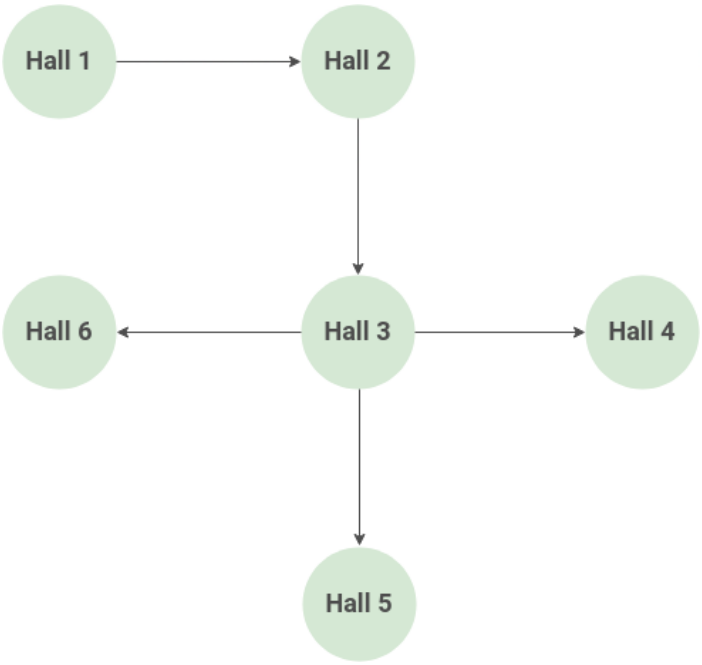
CSP

Artificial Intelligence Project 03

توضیح کلی پروژه.

هدف ما در این پروژه، مدل‌سازی یک مساله به صورت مساله‌ی ارضای محدودیت است. در مسائل ارضای محدودیت، یک مجموعه از متغیر ها داریم که هر کدام از متغیر ها میتوانند از دامنه‌ی مربوط به خود مقداری بگیرند. همچنین هر مساله یک مجموعه محدودیت دارد که هدف ما این است که طوری به این متغیر ها از دامنه‌شان مقدار دهیم که مجموعه محدودیت هایمان ارضا شوند و هیچ محدودیتی نقض نشود.
در مسائل ارضای محدودیت، راه حل، یک انتساب کامل و سازگار است. به این معنی که به همه‌ی متغیر ها باید مقدار دهیم که این همان کامل بودن راه حل است و مقداردهی باید به صورتی باشد که هیچ محدودیتی نقض نشود و این همان سازگاری است.
در این پروژه، ما تعدادی سالن و تعدادی گروه داریم که این گروه ها باید در سالن ها قرار بگیرند به نحوی که محدودیت های گفته شده را نقض نکنند.
برای مدل سازی مساله به مساله‌ی ارضای محدودیت، باید متغیر های مساله، دامنه‌ی مقادیر و محدودیت های مساله را مشخص کنیم. در مساله‌ای که ما با آن مواجه هستیم، متغیر ها، سالن ها هستند و مقادیری که باید به هر سالن انتساب داد، گروه ها هستند. برای مثال، مساله‌ی گفته شده در صورت پروژه را میتوان به فرم زیر به صورت مساله‌ی ارضای محدودیت مدل‌سازی کرد:



Variables = { Hall 1, Hall 2, Hall 3, Hall 4, Hall 5, Hall 6}

Values = { CE, ME, EE }

Constraints = adjacent halls must have different groups.

پس به طور کلی میتون گفت :

Variables = { i I $\forall i \ i \ \in \ N$ }

Values = { x I $\forall x \ x \ \in \ M$ }

E = adjacent halls must have different groups and values that we assign to variables, be in the preferences list.

الگوریتم ها

- **الگوریتم جستجوی عقب‌گرد(Backtracking)**

جستجوی عقب گرد همان جست و جوی عمقی (DFS) است که با عمال دو بهبود در آن الگوریتم، به جستجوی عقب‌گرد میرسیم. بهبود اول این است که برای انتساب مقادیر به متغیرها، ترتیبی در نظر بگیریم و محدودیت های نقض شده را در نظر بگیریم. برای مثال اگر تا مرحله‌ای پیش رفتیم و محدودیتی نقش شد، این تناقض را تشخیص داده و دیگر این مسیر را ادامه ندهیم و مسیر دیگری را امتحان کنیم که این روند همان الگوریتم عقب‌گرد است که یک جستجوی ناآگاهانه و پایه برای حل مسائل CSP است که در ادامه با استفاده از الگوریتم ها و روش هایی، سعی در بهبود این الگوریتم داریم.

ایده‌ی جستجوی عقب‌گرد به این صورت است که:

- در هر مرحله به یک متغیر مقدار میدهیم یعنی یک ترتیب خاص برای مقداردهی درنظر گرفتیم.
- در حین پیشروی، محدودیت ها را در نظر بگیریم و اگر در طی مسیر محدودیتی نقض شد، دیگر این مسیر را ادامه ندهیم.

- **بهبود الگوریتم جستجوی عقب‌گرد**

برای بهبود الگوریتم عقب‌گرد، دو ایده وجود دارد. ترتیب‌دهی (ordering) و فیلترکردن (filtering).

- **ترتیب‌دهی (ordering)**

این ایده به دو صورت اجرا می‌شود. یک حالت ترتیب‌دهی به متغیرهاست که به این صورت است که با چه ترتیبی به متغیرها مقدار دهیم تا اگر قرار است به شکست مواجه شویم، این شکست را زودتر بفهمیم.

حال فرض میکنیم یکی از متغیر ها راد انتخاب کردیم تا از بین مقادیر موجود در دامنه‌اش، به آن مقدار دهیم. حالت دومی که برای ترتیب‌دهی وجود دارد، ترتیب‌دهی به مقادیر است. یعنی کدام مقدار از دامنه‌ی متغیر را در اولویت قرار دهیم و به متغیر انتسابش دهیم تا اگر جوابی برای مساله وجود دارد، زودتر به جواب برسیم.

توابع هیوریستیک MRV و LCV ایده‌ی بهبود ترتیب‌دهی را اجرایی می‌کنند که در ادامه به توضیح این دو تابع می‌پردازیم.

- **تابع هیوریستیک MRV**

تابع هیوریستیک MRV یا Minimun Remaining Value برای انتخاب متغیری با کمترین تعداد مقادیر یا دامنه های ممکن باقی مانده استفاده می شود. در این مساله یعنی انتخاب گروهی که دارای کمترین سالن ترجیحی باقی مانده باشد. این تابع هیوریستیک را می توان با شمارش تعداد سالن های ترجیحی باقی مانده برای هر گروه اختصاص نیافته و انتخاب گروهی با کوچکترین تعداد محاسبه کرد.

پیاده سازی این تابع به شرح زیر است:

```
def MRV(halls: list[Hall]) -> Hall:
      min_remaining_values = maxsize
      selected_hall = None

      for hall in halls:
      if len(hall.getPrefrences()) < min_remaining_values and
hall.getValue() is None:
            min_remaining_values = len(hall.getPrefrences())
            selected_hall = hall

      return selected_hall
```

توضیح کد.

این کد یک تابع MRV را تعریف می کند که لیستی از سالن ها را به عنوان ورودی می گیرد و سالنی را با حداقل مقادیر باقی مانده (MRV) از لیست برمی گرداند. این تابع با مقداردهی اولیه دو متغیر، min_remaining_values با حداکثر مقدار ممکن و Select_hall با None شروع می شود. سپس برای هر سالن، بررسی می‌کند که آیا تعداد ترجیحات آن کمتر از حداقل مقادیر باقی‌مانده فعلی است و همچنین بررسی میکند که آیا مقداری به آن اختصاص داده نشده. اگر هر دو شرط برآورده شوند، مقدار min_remaining_values را با تعداد ترجیحات سالن فعلی به روز می کند و را با سالن فعلی را در select_hall میریزد.در نهایت سالن انتخاب شده را با MRV برمی گرداند.

- **تابع هیوریستیک LCV**

تابع هیوریستیک LCV یا Least Constraining Value یکی دیگر از توابع هیوریستیک در مسائل ارضای محدودیت است که هدف آن این است که مقادیر یک متغیر را مرتب کند تا تأثیر آن بر تعداد گزینه های باقی مانده برای همسایگان آن به حداقل برسد.

پیاده سازی این تابع به شرح زیر است:

```
def LCV(hall: Hall) -> list:
      """Least Constraining Value heuristic"""

      neighbors = hall.getNeighbors()
      prefrences = hall.getPrefrences()

      # sort the prefrences based on the number of remaining options of
nighbors
      sorted_prefrences = sorted(prefrences, key=lambda x: sum(1 for
neighbor in neighbors if x in neighbor.getPrefrences()))

      return sorted_prefrences
```

توضیح کد.

این تابع یک شی از کلاس Hall را به عنوان ورودی می گیرد که یک متغیر در مسئله ارضای محدودیت است. این تابع ابتدا همسایه های سالن ورودی را بازیابی می کند که متغیرهای دیگری هستند که مستقیماً به متغیر ورودی متصل هستند و سپس، ترجیحات مقادیر ممکن متغیر ورودی را بازیابی می کند. هدف این تابع LCV این است که این اولویت‌ها را بر اساس نحوه تأثیرگذاری آنها بر تعداد گزینه‌های باقی‌مانده همسایگان ترتیب دهد. برای انجام این کار، تابع لیستpreferences را با استفاده از تابع sorted و یک تابع lambda به عنوان آرگومان کلیدی مرتب می کند. تابع lambda تعداد همسایگانی را محاسبه می کند که هنوز preference فعلی را به عنوان یک گزینه دارند.

- **فیلتر کردن (filtering)**

ایده‌ی بعدی برای بهبود الگوریتم عقب‌گرد، فیلترکردن است که ایده‌ی این بهبود به این صورت است که مقادیر بی‌فایده را از دامنه‌ی مقادیری که هنوز به آنها مقداری انتساب ندادیم، حذف کنیم. یکی از روش های فیلتر کردن را در ادامه بررسی می‌کنیم.

- **الگوریتم بررسی رو به جلو (Forward Checking)**

این الگوریتم یکی از روش های فیلتر کردن است , تکنیکی است که در مسائل رضایت محدودیت (CSP) برای کاهش تعداد احتمالات آینده برای هر متغیر استفاده می شود.

ایده اصلی بررسی رو به جلو این است که مجموعه ای از مقادیر ممکن را برای هر متغیر حفظ کنیم و هر زمان که به یک متغیر مقداری اختصاص داده شد، مجموعه ها را به روز کنیم. این الگوریتم با بررسی محدودیت‌ها بین متغیرها و به روز رسانی دامنه هر متغیر بر اساس مقادیری که قبلاً به متغیرهای دیگر اختصاص داده شده است کار می کند. اگر دامنه یک متغیر خالی می شود، سپس تخصیص فعلی

نامعتبر در نظر گرفته می شود و الگوریتم به عقب برمی گردد تا مقدار دیگری را امتحان کند و به متغیر انتساب دهد.
بررسی رو به جلو می تواند به هرس فضای جستجوی مسائل ارضای محدودیت کمک کند و از بررسی شاخه های نامربوط یا بن بست درخت جستجو جلوگیری کند.

کد الگوریتم جستجوی عقب‌گرد و بررسی رو به جلو به شرح زیر است:

```
def FC(halls: list[Hall], index: int = 0, MRV = None, LCV = None, AC3=
None) -> ResponseModel:
      """forward checking algorithm"""
      halls = AC3(halls).result if AC3 is not None and not
AC3(halls).hasError else halls

      hall = MRV(halls) if MRV is not None else halls[index % len(halls)]

      index = halls.index(hall)

      # check if hall has prefrences to assign value
      if (len(hall.getPrefrences()) == 0):
      return ResponseModel([], True, f'No prefrences for {hall.getName()}')

      prefrences = LCV(hall) if LCV is not None else hall.getPrefrences()

      for prefrence in prefrences:
      copy_halls = deepcopy(halls)
      hall = copy_halls[index]

      hall.setValue(prefrence)

      # remove prefrence from nighbors
      for nighbor in hall.getNeighbors():
            nighbor.removePrefrence(prefrence)

      # all halls are checked
      if (Hall.checkAll(copy_halls)):
            return ResponseModel(copy_halls, False, 'forward checking
completed')

      # check if forward checking is completed successfully in the next
halls
```

```
      forward = FC(halls = copy_halls, index = halls.index(MRV(halls)),
MRV= MRV, LCV= LCV, AC3= AC3) if MRV is not None else FC(halls =
copy_halls, index= index + 1)

      if (not forward.hasError):
            copy_halls = forward.result

            return ResponseModel(copy_halls, False, 'forward checking
completed')
```

توضیح کد.

این تابع فهرستی از سالن‌ها را که متغیرهای موجود در مساله هستند را به همراه پارامترهای اختیاری ،MRV LCV و AC3 به عنوان ورودی دریافت می کند. این تابع با بررسی اینکه آیا الگوریتم AC3 ارائه شده است شروع می شود، و اگر چنین باشد، از الگوریتم AC3 برای کاهش دامنه متغیرها استفاده می کند. اگر الگوریتم AC3 ارائه نشود، از سالن های ورودی بدون کاهش دامنه استفاده می شود. در مرحله بعد، تابع سالنی را برای اختصاص یک مقدار انتخاب می کند. اگر الگوریتم MRV ارائه شده باشد، برای انتخاب سالن با حداقل مقادیر باقیمانده استفاده می شود سپس تابع بررسی می‌کند که آیا سالن انتخاب شده دارای اولویت‌هایی برای اختصاص دادن است یا خیر، اگر هیچ اولویتی نداشته باشد، تابع یک پیام خطا نشان می‌دهد. اگر سالن دارای ترجیحات باشد، تابع مقدار بعدی را برای تخصیص بر اساس الگوریتم LCV در صورت ارائه انتخاب می‌کند، در غیر این صورت، اولویت‌ها همانطور که هست استفاده می‌شوند.

الگوریتم بررسی رو به جلو، تمام شکست ها را نمیتواند تشخیص دهد. برای مثال یک مسیر ممکن است به جواب نرسد اما forward checking نمیتواند آن را تخیص دهد و الگوریتم را از ادامه‌ی مسیر بازدارد. راه حل این مشکل، استفاده از یکی دیگر از روش های فیلتر کردن است که در ادامه به بررسی آن میپردازیم.

- **الگوریتم سازگاری کمان (AC3)**

  الگوریتم AC3 یا Arc Consistency 3 یک الگوریتم ارضای محدودیت است که برای بهبود سازگاری متغیرها در یک مسئله CSP استفاده می شود. ایده پشت الگوریتم، حذف مقادیری است که نمی‌توان آنها را بر اساس محدودیت‌های بین متغیرها به یک متغیر نسبت داد. پیاده سازی این الگوریتم به شرح زیر است:

```
def AC3(halls: list[Hall]) -> ResponseModel:
      """AC3 algorithm"""
      queue: list[tuple(Hall, Hall)] = [(hall, neighbor) for hall in halls
for neighbor in hall.getNeighbors()]

      while queue:
      hall, neighbor = queue.pop(0)

      if revise(hall, neighbor):
            if len(hall.getPrefrences()) == 0:
                  return ResponseModel([], True, f'Contradiction detected
in AC3. No prefrences for {hall.getName()}')

            for n in hall.getNeighbors():
                  if n != neighbor:
                  queue.append((n, hall))

      return ResponseModel(halls, False, 'AC3 completed successfully')


def revise(hall: Hall, neighbor: Hall) -> bool:
      """check if hall and neighbor have a common value in their
preferences,
      if not remove the value from hall's preferences"""
      revised = False

      for value in hall.getPrefrences():
      if not any(prefrence.getName() != value.getName() for prefrence in
neighbor.getPrefrences()):
            hall.removePrefrence(value)
            revised = True

      return revised
```

توضیح کد.

الگوریتم AC3 با حفظ صفی از کمان‌ها کار می‌کند که هر کمان یک محدودیت بین دو متغیر را نشان می‌دهد. هر کمان درواقع یال های گراف هستند. الگوریتم با افزودن تمام کمان ها به صف و سپس بررسی مکرر سازگاری هر کمان با فراخوانی تابع "revise" شروع می شود. اگر یک ویرایش انجام شود، کمان های متصل به متغیر اصلاح شده به صف اضافه می شوند.

تابع "revise" بررسی می کند که آیا مقداری در دامنه یک متغیر وجود دارد که نمی توان مقداری را در دامنه متغیر

دیگر اختصاص داد. اگر چنین مقداری پیدا شود، از دامنه متغیر اول حذف می شود. این تابع یک boolean را برمی‌گرداند که نشان می‌دهد آیا ویرایش انجام شده است یا خیر. الگوریتم AC3 یک روش کارآمد برای بهبود ثبات در CSP است، به خصوص زمانی که تعداد محدودیت‌ها زیاد باشد. در الگوریتم های مختلف جستجو برای کوچکتر کردن فضای جستجو و مدیریت بیشتر استفاده می شود.

گراف محدودیت مساله.