به نام خدا

پروژه نهایی هوش مصنوعی : CSP

اعضای گروه:
حامد مسعودی
پگاه مطهری نژاد
مرتضی مصطفی زاده

فاز اول (مدلسازی مسئله) :

فرض کنیم n تعداد سالن های حوزه آزمون و m تعداد گروه های شرکت‌کننده باشد.

از آنجا که هر سالن تنها میتوند یک گروه شرکت‌کننده را شامل باشد پس در نظر میگیریم x_i گروه شرکت‌کننده در سالن i باشد؛ که چون گروه‌های مشخصی مانند مجموعه گروه‌های D_i علاقه‌مند به حضور در سالن i هستند لذا بایستی $x_i \in D_i$.

از طرفی بایستی به ازای هر دو سالن، اگر سالن اول به سالن دوم مسیر داشته باشد آنگاه گروه‌های شرکت‌کننده در این دو سالن با یکدیگر متفاوت باشد. یعنی اگر فرض کنیم E مجموعه تمام ارتباطات بین سالن ها باشد در این صورت داشته باشیم:

$$\forall\, i,j\,;(i,j) \in E \rightarrow x_i \neq x_j.$$

به طور معادل قید بالا را میتوان به صورت زیر بازنویسی کرد:

$$\forall\, i,j\,;\big((i,j) \in E \vee (j,i) \in E\big) \rightarrow x_i \neq x_j.$$

* بنابراین در عمل تفاوتی میان آنکه در نظر بگیریم سالن i به سالن j مسیر دارد با اینکه در نظر بگیریم بین سالن های i و j مسیر وجود دارد نیست. از این نکته در پیاده سازی برای کاهش محاسبات استفاده خواهیم نمود.

در نتیجه مسئله ارضای محدودیت به صورت زیر شکل میگیرد:

Variables: $x_i \;\; for\; i \in \{1,\dots,n\}$

Domains: $x_i \in D_i \; for\; i \in \{1,\dots,n\}$

Constraints: $\forall\, i,j \in \{1,\dots,n\}\,;\big((i,j) \in E \vee (j,i) \in E\big) \rightarrow x_i \neq x_j.$

برای پیاده سازی مدل مسئله از دو کلاس $Node$ و $Salon$ استفاده میکنیم.

```python
class Node:
    salons: list['Salon']
```

کلاس $Node$ برای ذخیره سازی یک وضعیت خاص از سالن‌ها استفاده می‌شود.

```python
class Salon:
    neighbors: list[int]
    domain: list[int]
    assigned: bool
```

کلاس $Salon$ برای ذخیره‌سازی همسایه‌های یک سالن و دامنه مجاز آن استفاده میشود که اگر $Salon.assigned$ مقدار درست باشد به معنی این است که آن سالن مقدار خود را دریافت کرده و دامنه آن نیز آن مقدار را ذخیره خواهد کرد.

فاز دوم (پیاده سازی الگوریتم backtracking) :

پیاده سازی backtracking :

```
def backtracking(root: Node, mode: int= 1) -> Node | None:
    stack = [(root, None, None)]

    while stack:
        state, salon, group = stack.pop()

        if isFailure(state):
            continue

        if isComplete(state):
            return state

        if mode == 0:
            if salon != None:
                forward_checking(state, salon, group)
        elif mode == 1:
            if salon != None:
                forward_checking(state, salon, group)
            else:
                if not AC3(state):
                    continue
        elif mode == 2:
            if not AC3(state):
                continue

        salon = MRV(state)
        for group in LCV(state, salon)[-1::-1]:
            if isSatisfy(state, salon, group):
                child = state.copy()
                child.salons[salon].domain = [group]
                child.salons[salon].assigned = True
                stack.append((child, salon, group))


    return None
```

ورودی این تابع یک وضعیت ابتدایی از سالن‌ها است؛ شامل لیستی از سالن هایی که هر کدام لیستی از همسایه ها و دامنهٔ خود را دربردارند. همچنین ***mode*** نشان دهنده حالات مختلف استفاده از الگوریتم ***AC3*** و ***forward checking*** را انتخاب میکند.

الگوریتم ***backtracking*** را با استفاده از ساختمان داده پشته پیاده کرده ایم که میتوان در بالا مشاهده کرد و توابع مورد نیاز در آن در ادامه آمده است.

پیاده سازی isFailure :

```
def isFailure(node: Node) -> bool:
    for salon in node.salons:
        if len(salon.domain) == 0:
            return True

    return False
```

این تابع نشان می‌دهد آیا وضعیت ورودی از سالن ها بدون جواب یا غیرممکن است یا خیر.

پیاده سازی isComplete :

```
def isComplete(node: Node) -> bool:
    for salon in node.salons:
        if not salon.assigned:
            return False

    return True
```

این تابع نشان می‌دهد آیا وضعیت ورودی یک جواب است یا خیر.

پیاده سازی isSatisfy :

```
def isSatisfy(node: Node, salon: int, group: int) -> bool:
    for neighbor in node.salons[salon].neighbors:
        neighbor = node.salons[neighbor]
        if neighbor.domain == [group]:
            return False

    return True
```

این تابع نشان می‌دهد آیا اگر گروه ورودی در سالن ورودی قرارگیرد (با این فرض که آن گروه علاقه‌مند به حضور در آن سالن است) در وضعیت ورودی قابل قبول است یا خیر.

پیاده سازی ForwardChecking :

```
def forward_checking(node: Node, salon: int, group: int) -> None:
    for neighbor in node.salons[salon].neighbors:
        neighbor = node.salons[neighbor]
        if group in neighbor.domain:
            neighbor.domain.remove(group)
```

این تابع برای انتشار محدودیت است و بعد از انتخاب یک مقدار برای سالن ، محدودیت را با سالن های مجاور بررسی می کند و دامنه سالن های مجاور را کاهش می دهد.

پیاده سازی MRV :

برای پیاده‌سازی MRV که هیوریستیکی برای مسئله CSP مطرح شده است، با توجه به الگوریتم MRV، نیاز است که سالنی را انتخاب کنیم که کمترین مقدار مجاز در دامنه را داشته باشد. زمانی که چندین سالن با یک اندازه دامنه وجود دارد سالنی را انتخاب می‌کنیم که محدودیت کمتری برای سالن‌های مجاور خود ایجاد کند.

```
def MRV(node: Node) -> int:
    max_neighbor = 0
    min_domain = sys.maxsize
    index = 0

    for i in range(len(node.salons)):
        if not node.salons[i].assigned:
            if len(node.salons[i].domain) < min_domain:
                min_domain = len(node.salons[i].domain)
                index = i

    for i in range(len(node.salons)):
        if len(node.salons[i].domain) == min_domain and not node.salons[i].assigned:
            if len(node.salons[i].neighbors) > max_neighbor:
                max_neighbor = len(node.salons[i].neighbors)
                index = i

    return index
```

به عنوان ورودی به این تابع مسئله را که شامل لیستی از سالن‌هاست میدهیم و خروجی، شماره سالنی انتخابی‌ست.

برای این کار در بین سالن‌هایی که تا کنون مقداردهی نشده‌اند، سالنی با کمترین مقدار مجاز(کوچک‌ترین دامنه) را پیدا می‌کنیم و در صورت وجود چند سالن با این ویژگی، سالنی را از بین آنها انتخاب می‌کنیم که بیشترین همسایگی (ایجاد محدودیت) را داشته باشد.

پیاده سازی LCV :

میدانیم که LCV نیز هیوریستیکی دیگری برای مسئله است. الگوریتم LCV به دنبال اولویت‌دهی مقادیر مجاز سالن ورودی‌ست که با ملاک ایجاد کمترین محدودیت برای دیگر سالن‌ها پیاده سازی می‌شود.

```
def LCV(node: Node, salon: int) -> list[int]:
    domain_sort = node.salons[salon].domain.copy()
    score = 0

    i = 0
    while i < len(domain_sort):
        d = domain_sort[i]
        for n in node.salons[salon].neighbors:
            neighbor = node.salons[n]
            if d in neighbor.domain:
                if len(neighbor.domain) == 1:
                    domain_sort.remove(d)
                    score = -1
                    i -= 1
                    break
                else : score += 1
        if score != -1:
            domain_sort[domain_sort.index(d)] = (score, d)
        score = 0
        i += 1

    domain_sort.sort(key = lambda x: x[0])
    for i in domain_sort:
        domain_sort[domain_sort.index(i)] = i[1]

    return domain_sort
```

این تابع مسئله و شماره سالن انتخاب شده در MRV را به عنوان ورودی دریافت و لیستی از مقادیر مجاز با ترتیب ایجاد محدودیت به صورت صعودی به عنوان خروجی می‌دهد.

در تابع LCV به ازای هر مقدار در دامنه سالن، وجود اون مقدار در دامنه سالن‌های مجاور چک می‌شود و در صورتی که اون مقدار در دامنه‌ی سالنی موجود باشد به score که در واقع تعداد محدودیت‌های ایجاد شده است اضافه میشود و بر همین اساس در لیستی مرتب شده به ما برگردانده می‌شود.

فاز سوم (پیاده سازی الگوریتم AC3) :

تابع Constraint Variable :

```
def constraint_varibale(node:Node) -> list[tuple['Salon', 'Salon']]:
    constraint=list()
    for salon in node.salons:
        for neighbor in salon.neighbors:
            neighbor = node.salons[neighbor]
            constraint.append((salon,neighbor))
    return constraint
```

این تابع یک لیست از تاپل سالن هایی که با هم در محدودیت هستند بر می گرداند.

```
def AC3(node: Node , queue:list=None) -> bool:
    if queue == None:
        queue = constraint_varibale(node)

    while queue:
        (salon_i,salon_j)= queue.pop(0)
        if remove_inconsistent_values(salon_i, salon_j):
            if len(salon_i.domain) == 0:
                return False

            for salon_k in salon_i.neighbors:
                salon_k = node.salons[salon_k]
                if (salon_k, salon_i) not in queue:
                    queue.append((salon_k, salon_i))

    return True
```

این تابع الگوریتم انتشار محدودیت AC3 است.

```
def remove_inconsistent_values(cell_i : Salon, cell_j : Salon) -> bool: #returns
true if a value is removed
    removed = False

    for value in cell_i.domain:
        # if not any([value != poss for poss in cell_j.domain]):
        if cell_j.domain == [value]:
            cell_i.domain.remove(value)
            removed = True

    return removed
```

در پیاده سازی AC3 از remove_inconsistent_values استفاده کردیم که عمل حذف کردن مقدار محدود کننده دامنه دو سالن در محدودیت باهم را انجام می دهد.