Amirkabir University of Technology
(Tehran Polytechnic)

گزارش 5: پیاده‌سازی جدول کلمات با استفاده از CSP

استاد: دکتر مهدی قطعی

نام دانشجو: مریم علیپور حاجی آقا

شماره دانشجویی: ۹۶۱۲۰۳۷

پیاده‌سازی جدول جستجوی کلمات با استفاده از CSP و Backtracking search

چکیده: بسیاری از مسائل موجود در هوش مصنوعی را می‌توان به عنوان مسائل ارضای محدودیت بررسی کرد از این رو توسعه تکنیک‌های راه حل موثر برای CSP ها یک مسئله مهم تحقیقاتی است.در این گزارش ما اولین الگوریتم های حل CSP را بررسی می کنیم و سپس یکی از مسائل قابل پیاده‌سازی با استفاده از الگوریتم‌های CSP را پیاده‌سازی می‌کنیم.

کلمات کلیدی: CSP، محدودیت، دامنه، backtracking search.

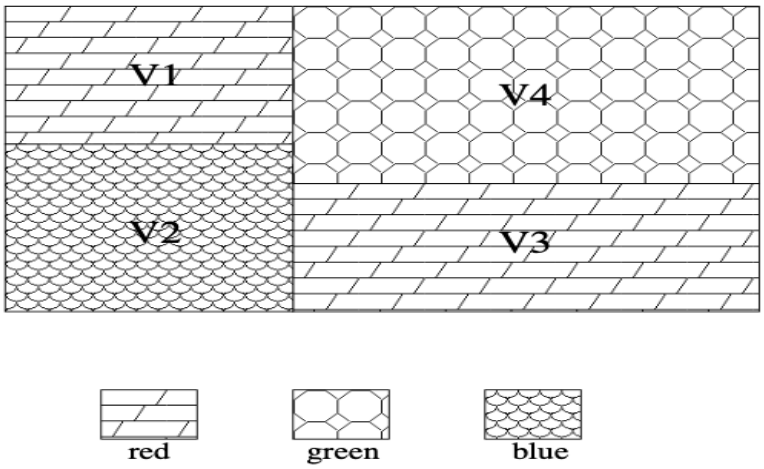
۱- مقدمه:

به طور کلی CSP مسئله‌ای است که متشکل از یک «مجموعه واحد متغیر» است که هر یک از آنها «یک دامنه ارزش واحد» و یک مجموعه محدودیت دارند. هر محدودیت برای برخی از زیر مجموعه‌های اصلی متغیرها تعریف می‌شود و مقادیری را که این متغیرها می‌توانند به طور همزمان بگیرند محدود می‌کند. وظیفه این است که یک مقدار برای هر متغیر به گونه ای تعیین شود که همه محدودیت‌ها را ارضا کند. در بعضی از مسائل هدف این است که چنین مقادیری را برای متغیرها تعیین کنیم.

۲- مسائل ارضای محدودیت :

مسئله N وزیر را می‌توان به صورت یک مسئله ارضای محدودیت مدل کرد. یک عدد صحیح N داده می‌شود و مسئله این است N وزیر در N خانه متفاوت در صفحه شطرنج طوری چیده شوند که محدودیت مسئله ارضا شود و محدودیت مسئله این است که هیچ کدام از دو وزیر یکدیگر را تهدید نکنند. دو وزیر زمانی یکدیگر را تهدید می‌کنند که در یک سطر یا ستون یا قطر قرار بگیرند.

مثال دیگر از CSP، مسئله رنگ آمیزی نقشه است. در این مسئله ما باید هر منطقه از نقشه را با یکی از مجموعه‌های مشخصی از رنگ‌ها رنگ کنیم، به طوری که هیچ دو منطقه مجاور از یک رنگ برخوردار نباشند.



به صورت کلی یک مسئله CSP می‌تواند به صورت یک تاپل سه تایی (V,D,R) تعریف شود به طوری که:

- V یک مجموعه‌ی n تایی از متغیرها است: $\{V_1, \dots, V_i, \dots, V_n\}$
- D یک مجموعه از دامنه‌ها است که برای هر i بین یک تا n داریم: $D_i = \{V_1^i, \dots, V_i^i, \dots, V_k^i\}$ یک مجموعه متناهی از مقادیر ممکن برای متغیر V_i است.
- R یک دنباله از روابط دودویی محدودیت‌ها است که برای هر $R_{ij} \in R$ داریم که محدودیت بین دو متغیر V_i و V_j را تعیین می‌کند.

۳- رویکردهای عمومی برای حل مسائل CSP:

Generate and Test: تمامی مقادیر ممکن برای همه متغیرها به صورت سیستماتیک جنریت می‌شوند و در محدودیت‌ها تست می‌شوند که آیا تمام محدودیت‌ها را ارضا می‌کنند یا خیر. اولین مقادیر یافت شده جواب مسئله است. اما در بدترین حالت (زمانی که می‌خواهیم تمام جواب‌های مسئله را پیدا کنیم) تعداد مقادیر یافت شده برای متغیرها برابر ضرب کارتزین دامنه‌های متغیرها است.

Tree Search: الگوریتم ابتدایی برای این جستجو Simple Backtracking است، یک استراتژی عمومی برای جستجو است که در حل بسیاری از مسائل کارآمد است.

در BT متغیرها یک به یک مقداردهی می‌شوند. وقتی متغیر مقدار دهی می‌شود، مقداری از دامنه آن انتخاب می‌شود و به آن اختصاص می‌یابد. سپس بررسی‌های محدودیت انجام می‌شود تا اطمینان حاصل شود که نمونه جدید با تمام قیدها سازگار است. تا کنون اگر تمام محدودیت‌های تکمیل شده برآورده شوند این متغیر با موفقیت نمونه سازی شده است و ما می‌توانیم به دنبال تغییر بعدی بعدی برویم. اگر محدودیت‌های خاصی را نقض کند در صورت موجود بودن یک مقدار جایگزین انتخاب می‌شود. اگر همه متغیرها یک مقدار داشته باشند توجه داشته

باشید که تمام واگذاری ها به طور کامل انجام می شوند و مشکل حل می‌شود. اگر در هر مرحله هیچ مقداری را نتوان به یک متغیر اختصاص داد بدون اینکه نقض محدودیت انجام شود Backtracking رخ می‌دهد.
این بدان معناست که جدیدترین متغیر نمونه، تجدیدنظر شده و مقدار جدیدی در صورت موجود بودن به آن اختصاص داده می شود. در این مرحله متغیرهای دیگر را مقداردهی می‌کنیم یا دوباره Backtracking انجام می‌دهیم. تا جایی ادامه می‌دهیم که راه حل پیدا شده است یا همه ترکیبی از نمونه سازی سعی شده است و شکست خورده است، این بدان معنی است که هیچ راه حلی وجود ندارد.
Constraint Propagation: انتشار محدودیت با هدف تبدیل یک CSP به یک مسئله معادل است که حل آن آسان‌تر است. کار در انتشار محدودیت با کاهش اندازه دامنه متغیرها به گونه ای که هیچ راه حلی رد نمی‌شود صورت می‌گیرد. این شامل حذف مقادیر اضافی از دامنه‌های متغیرها و انتشار نتایج این حذف در سراسر محدودیت‌ها است. انتشار محدودیت می تواند به درجات مختلف تغییر شکل دهد.

۴- نحوه پیاده‌سازی جدول کلمات با استفاده از CSP :

۴-۱- ابتدا یک فریم‌ورک برای CSP می‌سازیم:

محدودیت‌ها با استفاده از کلاس Constraint تعریف می‌شوند. هر Constraint شامل variables است که آن‌ها را محدود می‌کند و دارای تابع satisfied() است که بررسی می کند آیا آن محدودیت ارضا شده است یا خیر. تعیین اینکه آیا یک محدودیت ارضا شده است یا خیر، منطق اصلی است که برای تعریف یک مسئله خاص ارضای محدودیت را تعریف می کند. پیاده‌سازی توابع آن در زمانی اتفاق می‌افتد که در یک برنامه می‌خواهیم از این کلاس abstract استفاده کنیم. در واقع، این باید باشد، زیرا ما کلاس Constraint خود را به عنوان یک کلاس پایه انتزاعی تعریف می‌کنیم.

```
 6 V = TypeVar('V') # variable type
 7 D = TypeVar('D') # domain type
 8 # Base class for all constraints
 9 class Constraint(Generic[V, D], ABC):
10     # The variables that the constraint is between
11     def __init__(self, variables: List[V]) -> None:
12         self.variables = variables
13     # Must be overridden by subclasses
14     @abstractmethod
15     def satisfied(self, assignment: Dict[V, D]) -> bool:
16         ...
```

محور اصلی فریمورک CSP، کلاسی به نام CSP خواهد بود. CSP نقطه جمع آوری متغیرها ، دامنه‌ها و محدودیت‌ها است. از نظر نکات مربوط به نوع آن، از Generics استفاده می‌کند تا خود را به اندازه کافی انعطاف‌پذیر کند تا بتواند با هر نوع متغیر و مقادیر دامنه کار کند.مجموعه variables لیستی از متغیرها است، domains یک دیکشنری است که متغیرها را به لیست مقادیر ممکن برای آن مپ می‌کند و constraints یک دیکشنری است که هر متغیر را به لیستی از محدودیت‌ها که روی آن متغیر موثر هستند مپ می‌کند.

```
# A constraint satisfaction problem consists of variables of type V
# that have ranges of values known as domains of type D and constraints
# that determine whether a particular variable's domain selection is valid
class CSP(Generic[V, D]):
    def __init__(self, variables: List[V], domains: Dict[V, List[D]]) -> None:
        self.variables: List[V] = variables # variables to be constrained
        self.domains: Dict[V, List[D]] = domains # domain of each variable
        self.constraints: Dict[V, List[Constraint[V, D]]] = {}
        for variable in self.variables:
            self.constraints[variable] = []
            if variable not in self.domains:
                raise LookupError("Every variable should have a domain assigned to it.")
    def add_constraint(self, constraint: Constraint[V, D]) -> None:
        for variable in constraint.variables:
            if variable not in self.variables:
                raise LookupError("Variable in constraint not in CSP")
            else:
                self.constraints[variable].append(constraint)
```

چگونه می‌توان فهمید که یک پیکربندی داده شده از متغیرها و مقادیر دامنه انتخاب شده محدودیت‌ها را ارضا می‌کند؟ ما چنین پیکربندی داده شده را "assignment" می‌نامیم. ما به تابعی نیاز داریم که هر محدودیتی را برای یک متغیر معین نسبت به assignment بررسی کند تا ببیند آیا مقدار متغیر در assignment برای محدودیت‌ها کار می‌کند یا خیر. در اینجا ما تابع ()consistent را در CSP پیاده‌سازی می‌کنیم.

```
# Check if the value assignment is consistent by checking all constraints
# for the given variable against it
    def consistent(self, variable: V, assignment: Dict[V, D]) -> bool:
        for constraint in self.constraints[variable]:
            if not constraint.satisfied(assignment):
                return False
        return True
```

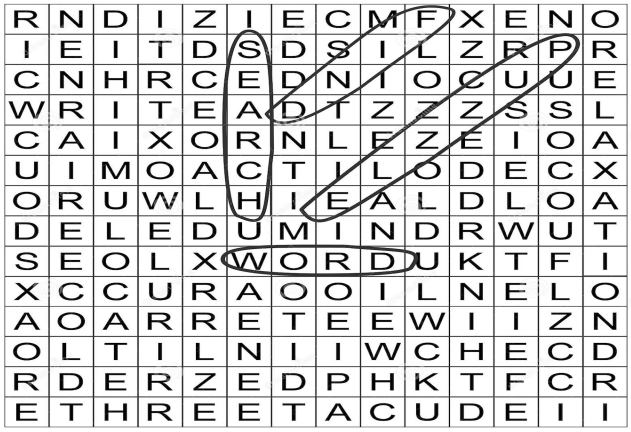
این فریمورک CSP از یک جستجوی simple backtracking برای یافتن راه حل برای مسئله استفاده خواهد کرد. Backtracking این ایده است که وقتی در جستجوی خود به دیواری برخورد کردید، به آخرین نقطه شناخته شده‌ای که قبل از دیوار تصمیم‌گیری کرده‌اید برگردید و مسیر دیگری را انتخاب کنید. اگر فکر می‌کنید به نظر می‌رسد DFS باشد، تقریبا درست فکر کردید. جستجوی backtrack که در تابع ()

backtracking_search زیر پیاده سازی شده است، نوعی جستجوی بازگشتی عمق اول است. این عملکرد به عنوان تابعی به کلاس CSP اضافه می‌شود.

```
49      def backtracking_search(self, assignment: Dict[V, D] = {}) -> Optional[Dict[V, D]]:
50          # assignment is complete if every variable is assigned (our base case)
51          if len(assignment) == len(self.variables):
52              return assignment
53          # get all variables in the CSP but not in the assignment
54          unassigned: List[V] = [v for v in self.variables if v not in assignment]
55          # get the every possible domain value of the first unassigned variable
56          first: V = unassigned[0]
57          for value in self.domains[first]:
58              local_assignment = assignment.copy()
59              local_assignment[first] = value
60              # if we're still consistent, we recurse (continue)
61              if self.consistent(first, local_assignment):
62                  result: Optional[Dict[V, D]] = self.backtracking_search(local_assignment)
63                  # if we didn't find the result, we will end up backtracking
64                  if result is not None:
65                      return result
66          return None
67
```

۲-۴- پیاده‌سازی جدول جستجوی کلمات با استفاده از فریم‌ورک CSP:

جستجوی کلمات در جدول، یک جدول با تعداد n در n حروف در هر خونه است که کلماتی در ستون‌ها و سطرها و قطرهای این جدول پنهان شده است. یک بازیکن از یک معمای جستجوی کلمات با اسکن دقیق از طریق جدول سعی در پیدا کردن کلمات پنهان دارد. یافتن مکان‌هایی برای قرار دادن کلمات به گونه‌ای که همه آن‌ها در جدول قرار بگیرند، نوعی مسئله ارضای محدودیت است. متغیرها کلمات هستند و دامنه‌ها مکان‌های احتمالی آن کلمات هستند. این مسئله در شکل زیر نشان داده شده است.



جستجوی کلمات ما شامل کلماتی نیست که با هم هم‌پوشانی داشته باشند.

ما شبکه را با حروف الفبای انگلیسی (ascii_uppercase) پر خواهیم کرد. برای نمایش شبکه به یک تابع نیز نیاز خواهیم داشت.

```
74 def generate_grid(rows: int, columns: int) -> Grid:
75 # initialize grid with random letters
76     return [[choice(ascii_uppercase) for c in range(columns)] for r in range(rows)]
77
78 def display_grid(grid: Grid) -> None:
79     for row in grid:
80         print("".join(row))
```

برای اینکه بفهمیم کلمات در کجای شبکه قرار می‌گیرند، دامنه‌های آن‌ها را تولید خواهیم کرد. دامنه یک کلمه لیستی از لیست مکان‌های احتمالی همه حروف آن است. کلمات نمی‌توانند در هر جایی قرار گیرند آنها باید در یک ردیف، ستون یا مورب قرار بگیرند که در محدوده شبکه است. به عبارت دیگ ، آنها نباید از انتهای شبکه خارج شوند. هدف از تولید generate-domain() ساخت این لیست‌ها برای هر کلمه است.

```
82 def generate_domain(word: str, grid: Grid) -> List[List[GridLocation]]:
83     domain: List[List[GridLocation]] = []
84     height: int = len(grid)
85     width: int = len(grid[0])
86     length: int = len(word)
87     for row in range(height):
88         for col in range(width):
89             columns: range = range(col, col + length + 1)
90             rows: range = range(row, row + length + 1)
91             if col + length <= width:
92                 # left to right
93                 domain.append([GridLocation(row, c) for c in columns])
94                 # diagonal towards bottom right
95                 if row + length <= height:
96                     domain.append([GridLocation(r, col + (r - row)) for r in rows])
97             if row + length <= height:
98                 # top to bottom
99                 domain.append([GridLocation(r, col) for r in rows])
100                # diagonal towards bottom left
101                if col - length >= 0:
102                    domain.append([GridLocation(r, col - (r - row)) for r in rows])
103    return domain
```

برای بررسی صحت یک راه حل بالقوه، باید یک محدودیت دست‌ساز را برای جستجوی کلمه اعمال کنیم.تابع satisfied() از WordSearchConstraint () به سادگی بررسی می‌کند که آیا هر یک از مکان‌های پیشنهادی برای یک کلمه همان مکانی است که برای کلمه دیگری پیشنهاد شده است یا خیر. این کار را با استفاده از set انجام می‌دهد. تبدیل list به set، همه موارد تکراری را حذف می‌کند. اگر موارد موجود در set کمتر از list اصلی باشد ، این بدان معنی است که لیست اصلی حاوی موارد تکراری بوده است. برای آماده سازی

داده‌ها برای این بررسی، ما از درک لیست تا حدودی پیچیده برای ترکیب چندین لیست فرعی از مکان برای هر کلمه در assignment در یک لیست بزرگتر از مکان استفاده خواهیم کرد.

```
105 class WordSearchConstraint(Constraint[str, List[GridLocation]]):
106     def __init__(self, words: List[str]) -> None:
107         super().__init__(words)
108         self.words: List[str] = words
109 
110     def satisfied(self, assignment: Dict[str, List[GridLocation]]) -> bool:
111         # if there are any duplicates grid locations then there is an overlap
112         all_locations = [locs for values in assignment.values() for locs in values]
113         return len(set(all_locations)) == len(all_locations)
```

سرانجام، ما آماده اجرای هستیم. برای این مثال، ما پنج کلمه را در یک شبکه نه در نه داریم. راه حلی که به دست ما می‌رسد باید شامل نگاشت بین هر کلمه و مکان‌هایی باشد که حروف آن در شبکه جای می‌گیرد.

```
115 if __name__ == "__main__":
116     grid: Grid = generate_grid(9, 9)
117     words: List[str] = ["MARYAM", "MOHAMMAD", "ALI", "SARAH", "FATEME"]
118     locations: Dict[str, List[List[GridLocation]]] = {}
119     for word in words:
120         locations[word] = generate_domain(word, grid)
121     csp: CSP[str, List[GridLocation]] = CSP(words, locations)
122     csp.add_constraint(WordSearchConstraint(words))
123     solution: Optional[Dict[str, List[GridLocation]]] = csp.backtracking_search()
124     if solution is None:
125         print("No solution found!")
126     else:
127         for word, grid_locations in solution.items():
128             # random reverse half the time
129             if choice([True, False]):
130                 grid_locations.reverse()
131             for index, letter in enumerate(word):
132                 (row, col) = (grid_locations[index].row, grid_locations[index].column)
133                 grid[row][col] = letter
134         display_grid(grid)
135 
```

و در نهایت جدول کلمات تولید شده به صورت زیر است:

```
To update your account to use zsh, please run `chsh -s /bin/zsh`.
For more details, please visit https://support.apple.com/kb/HT208050.
Maryams-MacBook-Air:~ maryamalipour$ /Library/Frameworks/Python.framework/Versions/3.8/bin/
HMAYRAMMA
GHARASTOL
IWBXNTEHI
MNOUHFMAH
LIWBBHEMR
FPKNBCTMH
SHMYFTAAW
XNQTNWFDY
WVQEVFHYX
Maryams-MacBook-Air:~ maryamalipour$
```

۵- نتیجه‌گیری:

تبدیل کردن مسائل به CSP یکی از روش‌های راحت و جالب برای حل مسئله است و از آن جایی که الگوریتم‌های متفاوت با پیچیدگی زمانی متفاوت و پیچیدگی فضای متفاوت برای حل مسائل CSP وجود دارد برای تعداد خوبی از محققان عملیات روش به صرفه‌ای برای مشکلاتشان است

منابع:

http://www.cs.columbia.edu/~evs/ais/finalprojs/steinthal/

https://www.sciencedirect.com/science/article/abs/pii/S0377221798003646

https://en.wikipedia.org/wiki/Constraint_satisfaction_problem

https://github.com/aimacode/aima-python/blob/master/csp.py

https://livebook.manning.com/book/classic-computer-science-problems-in-python/chapter-3/v-3/67