دانشگاه صنعتی امیرکبیر

(پلی‌تکنیک تهران)

دانشکده ریاضی و علوم کامپیوتر

درس هوش مصنوعی و کارگاه

گزارش ۴: پیاده‌سازی بازی Kakuro با روش‌های ارضای محدودیت

نگارش

مهسا گودرزی

۹۹۱۲۰۴۳

استاد اول

دکتر مهدی قطعی

استاد دوم

بهنام یوسفی مهر

آذر ۱۴۰۲

چکیده

الگوریتم‌های حل مسائل ارضای محدودیت در هوش مصنوعی، روش‌های مختلفی برای یافتن مقادیری برای متغیرها هستند که مجموعه‌ای از محدودیت‌ها را ارضا می‌کنند. از الگوریتم‌های رایج برای حل این دسته از مسائل، می‌توان به روش پس‌گرد اشاره کرد که این الگوریتم یک جستجوی عمق اول است که فضای جستجو را به صورت سیستماتیک بررسی می‌کند. در این مقاله به شرح کد بازی Kakuro پرداخته‌ایم که در این برنامه ابتدا عامل ساده‌ای برای حل این پازل با استفاده از روش پس‌گرد طراحی کرده‌ایم و سپس با استفاده از دیگر روش‌های ارضای محدودیت این عامل را هوشمند کرده و کارآیی آن را بهبود بخشیده‌ایم.

واژه‌های کلیدی:

مسائل ارضای محدودیت، روش پس‌گرد، Kakuro

فهرست مطالب

صفحه

چکیده ..... أ
فصل اول مقدمه ۱
فصل دوم پیاده‌سازی Kakuro با روش پس‌گرد ۵
۲-۱- نوع تعریف صفحه بازی ۶
۲-۲- توابع کمکی ۷
۲-۲-۱- تابع find_horizontal_cells ۷
۲-۲-۲- تابع find_vertical_cells ۸
۲-۲-۳- تابع find_row_clue ۹
۲-۲-۴- تابع find_col_clue ۱۰
۲-۲-۵- تابع play_kakuro_backtracking ۱۰
۲-۳- الگوریتم پس‌گرد و توابع مربوط به آن ۱۱
۲-۳-۱- تابع check_constraints ۱۲
۲-۳-۲- تابع solve_kakuro_backtracking ۱۳
۲-۴- نمونه اجرای بازی ۱۵
فصل سوم بهبود روش پس‌گرد ۲۰
۳-۱- توابع کمکی ۲۱
۳-۱-۱- تابع find_combinations_permutations ۲۱
۳-۱-۲- تابع find_common_numbers ۲۲
۳-۱-۳- تابع sort_cells_by_constraint ۲۲
۳-۱-۴- تابع play_kakuro_csp ۲۳
۳-۲- تابع solve_kakuro_csp ۲۴
۳-۳- نمونه اجرای بازی ۲۵
فصل چهارم لینک کد بازی Kakuro ۲۹
فصل پنجم جمع‌بندی و نتیجه‌گیری ۳۱
منابع ۳۳

فهرست اشکال

صفحه

شکل ۱-۱- نمونه‌ای از پازل Kakuro ۴
شکل ۲-۱- نمونه‌ای از نوع تعریف صفحه پازل در برنامه ۶
شکل ۲-۲- نمایش نمونه پازل شکل ۲-۱ در کنسول ۷
شکل ۲-۳- تصویر کد تابع print_board ۷
شکل ۲-۴- تصویر کد تابع find_horizontal_cells ۸
شکل ۲-۵- تصویر کد تابع find_vertical_cells ۹
شکل ۲-۶- تصویر کد تابع find_row_clue ۹
شکل ۲-۷- تصویر کد تابع find_col_clue ۱۰
شکل ۲-۸- تصویر کد تابع play_kakuro_backtracking ۱۱
شکل ۲-۹- کد تابع check_constraints ۱۳
شکل ۲-۱۰- کد تابع solve_kakuro_backtracking ۱۴
شکل ۲-۱۱- یک نمونه پازل Easy Kakuro ۱۵
شکل ۲-۱۲- حل نمونه پازل Easy Kakuro توسط عامل ساده ۱۵
شکل ۲-۱۳- یک نمونه پازل Medium Kakuro ۱۶
شکل ۲-۱۴- حل نمونه پازل Medium Kakuro توسط عامل ساده ۱۶
شکل ۲-۱۵- یک نمونه پازل Hard Kakuro ۱۷
شکل ۲-۱۶- حل نمونه پازل Hard Kakuro توسط عامل ساده ۱۷
شکل ۲-۱۷- یک نمونه پازل Expert Kakuro ۱۸
شکل ۲-۱۸- حل نمونه پازل Expert Kakuro توسط عامل ساده ۱۸
شکل ۳-۱- تصویر کد تابع find_combinations_permutations ۲۱
شکل ۳-۲- تصویر کد تابع find_common_numbers ۲۲
شکل ۳-۳- تصویر کد تابع sort_cells_by_constraint ۲۳
شکل ۳-۴- تصویر کد تابع play_kakuro_csp ۲۴

شکل ۳-۵- تصویر کد تابع solve_kakuro_csp ۲۵
شکل ۳-۶- حل نمونه پازل Easy Kakuro توسط عامل هوشمند ۲۶
شکل ۳-۷- حل نمونه پازل Medium Kakuro توسط عامل هوشمند ۲۶
شکل ۳-۸- حل نمونه پازل Hard Kakuro توسط عامل هوشمند ۲۷
شکل ۳-۹- حل نمونه پازل Expert Kakuro توسط عامل هوشمند ۲۷

فصل اول

مقدمه

مقدمه

هدف از مسائل رضایت محدودیت[۱] (CSPs)، که دسته‌ای از مسائل هوش مصنوعی است، یافتن راه‌حلی است که مجموعه‌ای از محدودیت‌ها را برآورده کند. به عبارت دیگر هدف این نوع از مسائل، یافتن مقادیری برای گروهی از متغیرها است که مجموعه‌ای از محدودیت‌ها یا قوانین را انجام دهند. سه جزء اصلی در CSP عبارتند از:

- متغیرها[۲]: عناصری هستند که باید مقادیری به آن‌ها اختصاص داده شود تا محدودیت‌های مشخصی را برآورده کنند.
- دامنه‌ها[۳]: مجموعه‌ای از مقادیر ممکن برای هر متغیر هستند.
- محدودیت‌ها[۴]: قوانینی هستند که تعیین می‌کنند متغیرها چگونه نسبت به یکدیگر قرار می‌گیرند.

یکی از الگوریتم‌های معروف برای حل CSP، الگوریتم پس‌گرد[۵] است که یک الگوریتم جستجوی عمق اول[۶] می‌باشد و به طور منظم فضای جستجوی راه‌حل‌های ممکن را تا زمانی که یک راه‌حل پیدا شود که تمام محدودیت‌ها را برآورده کند، بررسی می‌کند. این الگوریتم با انتخاب یک متغیر و اختصاص دادن یک مقدار به آن شروع می‌شود و سپس به طور مکرر تلاش می‌کند تا مقادیر را به سایر متغیرها اختصاص دهد. اگر در هر زمانی یک متغیر نتواند مقداری را که محدودیت‌ها را برآورده کند، پیدا کند، الگوریتم به متغیر قبلی بازمی‌گردد و مقدار دیگری را امتحان می‌کند.

---

۱ Constraint Satisfaction Problems

۲ Variables

۳ Domains

۴ Constraints

۵ Backtracking

۶ Depth First Search

از دیگر الگوریتم‌های رایج برای حل CSP، می‌توان به الگوریتم جستجوی رو به جلو[۷]، سازگاری کمان[۸]، MRV[۹]، LCV[۱۰] و... اشاره کرد. هر کدام از این الگوریتم‌ها به صورت خلاصه در ادامه شرح داده شده‌اند.

الگوریتم جستجوی رو به جلو پس از اختصاص دادن یک مقدار به یک متغیر، دامنه‌های متغیرهای بعدی را بررسی می‌کند و مقادیری که دیگر نمی‌توانند اختصاص داده شوند را حذف می‌کند. این کار باعث می‌شود که تعداد مقادیری که باید در مراحل بعدی بررسی شوند، کاهش یابد.

الگوریتم سازگاری کمان بررسی می‌کند که آیا برای هر جفت متغیر، مقداری در دامنه‌هایشان وجود دارد که با همه محدودیت‌های بین آن دو متغیر سازگار باشد. اگر مقداری وجود نداشته باشد، آن مقدار از دامنه حذف می‌شود.

الگوریتم MRV به انتخاب متغیری می‌پردازد که کمترین تعداد مقادیر باقی‌مانده را دارد. این کار باعث می‌شود که احتمال برخورد با محدودیت‌ها و بن‌بست‌ها در ابتدای جستجو بیشتر شود.

الگوریتم LCV به انتخاب مقداری می‌پردازد که کمترین محدودیت را برای متغیرهای دیگر ایجاد می‌کند. به این ترتیب، احتمال اینکه در مراحل بعدی به بن‌بست برسیم کاهش می‌یابد.

در این مقاله به بررسی عامل هوشمند حل کننده پازل Kakuro می‌پردازیم که این عامل ابتدا با استفاده از الگوریتم Backtracking پیاده‌سازی و سپس با استفاده از دیگر روش‌های حل CSP به بهبود آن پرداخته شده است.

در پازل Kakuro جمع اعداد مربوط به بخشی از یک سطر یا ستون به ما داده شده است و باید ارقام ۱ تا ۹ را به گونه ای در هر بخش قرار دهیم که جمع آنها با جمع مطلوب آن بخش برابر باشد. جمع مطلوب هر بخش را راهنما[۱۱] می‌گویند. شرط دیگری که باید رعایت شود این است که ارقامی که با هم جمع

[۷] Forward Checking

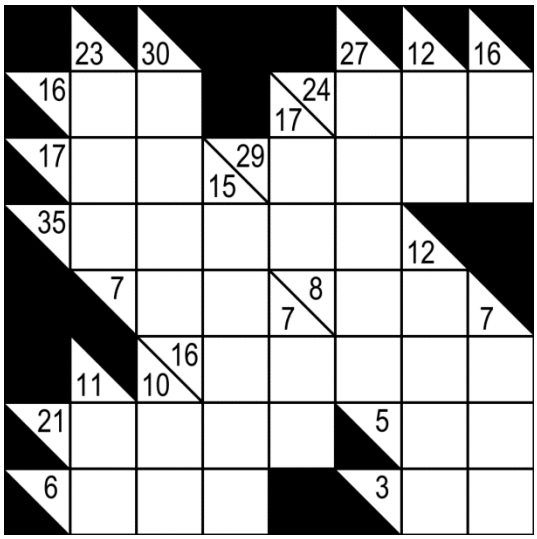
[۸] Arc Consistency

[۹] Minimum Remaining Values

[۱۰] Least Constraining Value

[۱۱] Clue

می‌شوند تا به یک راهنما برسند نباید تکراری باشند. خانه‌های خالی با رنگ سفید و خانه‌های راهنما با رنگ سیاه نمایش داده می‌شوند. در شکل ۱-۱ یک نمونه از این پازل نشان داده شده است.



شکل ۱-۱- نمونه‌ای از پازل Kakuro

بازی Kakuro را می‌توان به عنوان یک مسئله ارضای محدودیت مدل سازی کرد. چگونگی تعریف این پازل به عنوان یک CSP به صورت زیر است:

متغیرها: هر خانه سفید در پازل Kakuro یک متغیر است. هر متغیر باید یک عدد از ۱ تا ۹ را به خود اختصاص دهد.

دامنه‌ها: دامنه برای هر متغیر، مجموعه اعداد از ۱ تا ۹ است. این بدان معناست که هر خانه سفید می‌تواند هر یک از این اعداد را در خود جای دهد.

محدودیت‌ها: محدودیت‌ها در Kakuro بر اساس راهنماهای ارائه شده در خانه‌های سیاه تعریف می‌شوند. هر راهنمای عمودی یا افقی نشان‌دهنده مجموع اعدادی است که در خانه‌های سفید متصل به آن راهنما قرار می‌گیرند. علاوه بر این، هر سری از اعداد در یک خط یا ستون باید منحصر به فرد باشند.

فصل دوم

پیاده‌سازی Kakuro با روش پس‌گرد

پیاده‌سازی Kakuro با روش پس‌گرد

در این فصل کد عامل ساده بازی Kakuro که با استفاده از روش Backtracking نوشته شده است، به صورت خلاصه توضیح و علت تعریف تابع‌های مختلف شرح داده می‌شود. کد این بازی در Pycharm که یک محیط توسعه یکپارچه[1] برنامه نویسی پایتون می‌باشد، پیاده‌سازی شده است.

۲-۱- نوع تعریف صفحه بازی

محیط این بازی کاملا ساده است و تمامی جزئیات مربوط به بازی در کنسول چاپ می‌شود. اگر قصد حل پازلی را توسط عامل هوشمند داریم، باید در خود کد برنامه آن پازل را تعریف کنیم. برای انجام این کار، ما باید هر پازل را به صورت یک متغیر لیست تو در تو در نظر بگیریم، که هر کدام از این لیست‌های داخلی، نشان دهنده‌ی یک سطر از صفحه بازی است. خانه‌های سفید که در ابتدای بازی خالی هستند با " "، خانه‌های سیاه بدون راهنما با " x "، خانه‌های سیاه که راهنما داشته باشند نیز به سه حالت " x\n " یا " n\x " یا " n\m " (n و m اعداد دلخواه برای آن راهنما هستند) تعریف می‌شوند که هر کدام از این حالت‌ها به این بستگی دارد که آن خانه چند راهنما و در چه جهتی داشته باشد. برای مثال در " n\m " n نشان دهنده‌ی راهنمای ستون پایین خود و m نشان دهنده‌ی راهنمای سطر راست خود است. در شکل ۲-۱ نمونه‌ای از نوع تعریف صفحه پازل در کد بازی و در شکل ۲-۲ نمایش این صفحه در کنسول آورده شده است.

```
example_board = [
    ["  x  ", "  x  ", "26\\x ", "13\\x ", "  x  "],
    ["  x  ", "11\\10", "     ", "     ", "16\\x "],
    [" x\\13", "     ", "     ", "     ", "     "],
    [" x\\28", "     ", "     ", "     ", "     "],
    ["  x  ", " x\\15", "     ", "     ", "  x  "]
]
```

شکل ۲-۱- نمونه‌ای از نوع تعریف صفحه پازل در برنامه

[1] Integrated Development Environment (IDE)

```
|   x    |   x    | 26\x   | 13\x   |   x    |
|   x    | 11\10  |        |        | 16\x   |
|  x\13  |        |        |        |        |
|  x\28  |        |        |        |        |
|   x    |  x\15  |        |        |   x    |
```

شکل ۲-۲- نمایش نمونه پازل شکل ۲-۱ در کنسول

چاپ صفحه پازل در کنسول، به کمک تابع print_board انجام می‌پذیرد که تصویر کد این تابع در شکل ۲-۳ آورده شده است. این تابع به عنوان ورودی صفحه فعلی بازی که یک متغیر لیست است را دریافت و سپس جزئیات این صفحه را چاپ می‌کند.

```python
def print_board(board: list):
    for row in board:
        print(" | " + " | ".join(row) + " | ")
```

شکل ۲-۳- تصویر کد تابع print_board

۲-۲- توابع کمکی

در این برنامه، علاوه بر توابع اصلی مربوط به الگوریتم پس‌گرد، توابع کمکی دیگری نیز تعریف شده‌اند که به کمک آن‌ها عامل می‌تواند درباره‌ی جزئیات صفحه پازل اطلاعات لازم را جمع‌آوری کند. بعضی از این توابع هم در کد عامل ساده و هم در کد عامل هوشمند به کار رفته‌اند.

۲-۲-۱- تابع find_horizontal_cells

این تابع که تصویر کد آن در شکل ۲-۴ آورده شده است، خانه‌های سفید هم سطر با خانه‌ی ورودی به تابع را پیدا می‌کند و مقدار فعلی داخل این خانه‌ها را به صورت یک لیست برمی‌گرداند. ورودی‌های این تابع وضعیت فعلی صفحه‌ی بازی و شماره سطر و ستون خانه انتخابی می‌باشد. به کمک این تابع عامل می‌تواند تشخیص دهد که خانه‌های سفید هم سطر با خانه‌ی خالی مد نظر در این مرحله چه مقادیری در خود دارند.

```python
def find_horizontal_cells(board: list, row: int, col: int) -> list:
    horizontal_cells = [board[row][col]]

    for i in range(col - 1, -1, -1):
        cell = board[row][i]
        if "\\" in cell or "x" in cell:
            break
        horizontal_cells.append(cell)

    for i in range(col + 1, len(board)):
        cell = board[row][i]
        if "\\" in cell or "x" in cell:
            break
        horizontal_cells.append(cell)

    return horizontal_cells
```

شکل ۲-۴- تصویر کد تابع find_horizontal_cells

۲-۲-۲- تابع find_vertical_cells

عملکرد این تابع مشابه تابع find_horizontal_cells است، با این تفاوت که به جای خانه‌های سفید هم سطر، خانه‌های سفید هم ستون با خانه انتخابی را پیدا می‌کند و مقدار فعلی داخل آن‌ها را به صورت لیست برمی‌گرداند. تصویر کد این تابع در شکل ۲-۵ آورده شده است.

```python
def find_vertical_cells(board: list, row: int, col: int) -> list:
    vertical_cells = [board[row][col]]

    for i in range(row - 1, -1, -1):
        cell = board[i][col]
        if "\\" in cell or "x" in cell:
            break
        vertical_cells.append(cell)

    for i in range(row + 1, len(board)):
        cell = board[i][col]
        if "\\" in cell or "x" in cell:
            break
        vertical_cells.append(cell)

    return vertical_cells
```

شکل ۲-۵- تصویر کد تابع find_vertical_cells

۲-۲-۳- تابع find_row_clue

این تابع با دریافت وضعیت فعلی صفحه‌ی بازی و شماره سطر و ستون یک خانه سفید، می‌تواند راهنمای سطری این خانه که در سمت چپ آن قرار دارد را پیدا کند و عدد داخل این راهنما را به عنوان خروجی برگرداند. تصویر کد این تابع در شکل ۲-۶ آورده شده است.

```python
def find_row_clue(board: list, row: int, col: int) -> int:
    row_clue = 0

    for i in range(col - 1, -1, -1):
        cell = board[row][i]
        if "\\" in cell:
            if "x" != cell.split("\\")[1].strip():
                row_clue = int(cell.split("\\")[1].strip())
                break
        if "x" in cell:
            break

    return row_clue
```

شکل ۲-۶- تصویر کد تابع find_row_clue

۲-۲-٤- تابع find_col_clue

این تابع که تصویر کد آن در شکل ۲-۷ آورده شده است، عملکردی مشابه با تابع find_row_clue دارد، با این تفاوت که به جای راهنمای سطری، راهنمای ستونی یک خانه سفید که در سمت بالای آن قرار دارد را پیدا می‌کند و عدد داخل این راهنما را به عنوان خروجی برمی‌گرداند.

```python
def find_col_clue(board: list, row: int, col: int) -> int:
    col_clue = 0

    for i in range(row - 1, -1, -1):
        cell = board[i][col]
        if "\\" in cell:
            if "x" != cell.split("\\")[0].strip():
                col_clue = int(cell.split("\\")[0].strip())
                break
        if "x" in cell:
            break

    return col_clue
```

شکل ۲-۷- تصویر کد تابع find_col_clue

۲-۲-۵- تابع play_kakuro_backtracking

این تابع، تابع اجرای بازی و فراخوانی تابع اصلی مربوط حل کننده پازل می‌باشد. ورودی این تابع که تصویر کد آن در شکل ۲-۸ آورده شده است، مشخصات صفحه یک پازل می‌باشد که قصد حل آن را داریم، برای مثال اگر بخواهیم پازل نمونه شکل ۲-۱ را حل کنیم، در هنگام فراخوانی تابع play_kakuro_backtracking، به عنوان ورودی باید example_board را وارد کنیم. این تابع همچنین با دستور datetime.now() از کتابخانه datetime زمان شروع و پایان حل پازل توسط عامل را محاسبه می‌کند و سپس تفاضل این دو زمان را به عنوان زمان اجرای برنامه در پایان بازی چاپ می‌کند. در ابتدای این تابع، صفحه اصلی و اولیه پازل چاپ می‌شود، سپس با فراخوانی تابع حل کننده پازل، اگر عامل بتواند جوابی پیدا کند پیام پیدا شدن راه حل و صفحه‌ی نهایی پازل را چاپ می‌کند و اگر عامل نتواند جوابی برای حل پازل پیدا کند، پیام عدم وجود جواب را چاپ می‌کند. همچنین در آخر

تعداد مراحلی که عامل برای حل پازل طی کرده اســت و با متغیر level نمایش داده می‌شــود را نیز بر روی صـفحه چاپ می‌کند. در واقع level نشـان دهنده‌ی تعداد مراحلی اسـت که صـفحه‌ی بازی توسـط عامل تغییر کرده است و هر بار که صفحه‌ی جدیدی در کنسول چاپ شود، مقدار level نیز به اندازه یک واحد افزایش پیدا می‌کند.

```python
def play_kakuro_backtracking(board: list):
    start_time = datetime.now()

    print_board(board)
    print()

    if solve_kakuro_backtracking(board):
        print("🔥 Yeah!!! Kakuro puzzle solved! 🔥")
        print_board(board)
    else:
        print("☹️ Oh No! I couldn't find a solution. ☹️")

    end_time = datetime.now()
    execution_time = end_time - start_time

    print(f"Execution Time: {execution_time}")
    print(f"Number of Levels: {level}")
```

شکل ۲-۸- تصویر کد تابع play_kakuro_backtracking

۲-۳- الگوریتم پس‌گرد و توابع مربوط به آن

همان‌طور که گفته شــد، در ابتدا برای این پازل یک عامل ســاده طراحی کردیم که الگوریتم اصــلی آن، backtracking است. برای پیاده‌سازی این الگوریتم دو تابع اصلی داریم؛ اول تابع check_constraints و دوم تابع solve_kakuro_backtracking که در ادامه عملکرد و ســاختار هر کدام از این توابع را توضیح می‌دهیم.

۲-۳-۱- تابع check_constraints

این تابع محدودیت‌های بازی Kakuro را برای خانه‌ی سفید انتخاب شده با یک عضو از دامنه بررسی می‌کند. در واقع این تابع وضعیت فعلی صفحه بازی، شماره سطر و ستون خانه‌ی سفید مد نظر و یک عدد را به عنوان ورودی دریافت می‌کند، که این عدد از دامنه مقادیر ممکن برای هر متغیر یعنی اعداد ۱ تا ۹ انتخاب شده است. سپس تابع هر کدام از محدودیت‌ها را با توجه به این ورودی‌ها بررسی می‌کند و در صورتی که هر کدام از این محدودیت‌ها نقض شوند، عدد انتخابی از دامنه نمی‌تواند مقدار مناسب و درستی برای متغیر باشد و تابع مقدار False را برمی‌گرداند. همچنین اگر تمامی محدودیت‌ها ارضا شوند، تابع مقدار True را برمی‌گرداند.

در این تابع از چهار تابع اول معرفی شده در بخش ۲-۱ استفاده شده است که هر کدام از این تابع‌ها بسته به نوع عملکرد خود در تابع check_constraints نقش دارند. محدودیت اولی که در این تابع مورد بررسی قرار می‌گیرد این است که آیا عدد انتخابی از قبل در خانه‌های هم سطر یا هم ستون با خانه‌ی سفید مورد نظر وجود داشته است یا خیر. اگر اینطور باشد، یعنی با قرار دادن این عدد در این خانه، محدودیت عدم وجود عدد تکراری در هر سطر یا ستون نقض می‌شود بنابراین تابع مقدار False را برمی‌گرداند.

در ادامه، این تابع محدودیت بعدی را مورد بررسی قرار می‌دهد. یعنی بررسی می‌کند آیا با اضافه کردن عدد انتخابی به خانه‌ی سفید مد نظر، مجموع اعداد موجود در آن سطر یا ستون با راهنمای نظیر خود نابرابر می‌شود یا نه. به صورت دقیق‌تر، اگر بدون توجه به تعداد خانه‌های سفید خالی آن سطر یا ستون این مجموع از عدد راهنما بیشتر باشد، یا اگر تمام خانه‌های دیگر آن سطر یا ستون مقداردهی شده باشند و این مجموع از عدد راهنما کمتر باشد، آنگاه تابع مقدار False را برمی‌گرداند.

در انتها اگر هیچ کدام از محدودیت‌ها نقض نشوند، تابع مقدار True را برمی‌گرداند و به این معنی است که این عدد برای خانه‌ی سفید مد نظر می‌تواند مقدار مناسبی باشد. تصویر کد این تابع در شکل ۲-۹ آورده شده است.

```python
def check_constraints(board: list, row: int, col: int, num: int) -> bool:
    horizontal_cells = find_horizontal_cells(board, row, col)
    vertical_cells = find_vertical_cells(board, row, col)

    if f"  {num}  " in horizontal_cells:
        return False
    if f"  {num}  " in vertical_cells:
        return False

    row_clue = find_row_clue(board, row, col)
    col_clue = find_col_clue(board, row, col)

    empty_row_cells = horizontal_cells.count("     ")
    sum_row_constraint = sum([int(cell) for cell in horizontal_cells if cell != "     "]) + num

    if sum_row_constraint > row_clue:
        return False
    if sum_row_constraint < row_clue and empty_row_cells <= 1:
        return False

    empty_col_cells = vertical_cells.count("     ")
    sum_col_constraint = sum([int(cell) for cell in vertical_cells if cell != "     "]) + num

    if sum_col_constraint > col_clue:
        return False
    if sum_col_constraint < col_clue and empty_col_cells <= 1:
        return False

    return True
```

شکل ۲-۹- کد تابع check_constraints

۲-۳-۲- تابع solve_kakuro_backtracking

این تابع، تابع اصلی و پایه‌ی الگوریتم backtracking می‌باشد و یک تابع بازگشتی می‌باشد. مقادیر ورودی این تابع وضعیت فعلی صفحه‌ی بازی و شماره سطر و ستون خانه‌ای که عامل در این مرحله در آن قرار گرفته است می‌باشد. با توجه به بازگشتی بودن تابع، شرط پایه برای آن این است که عامل دیگر به انتهای پازل یعنی آخرین خانه صفحه‌ی بازی رسیده باشد و در صورتی که عامل تا به این مرحله پیش رفته باشد، یعنی یک راه حل برای پازل پیدا کرده است و بنابراین مقدار True برگردانده می‌شود. با توجه به اینکه عامل خانه‌ها را به ترتیب طی می‌کند، اگر به آخرین خانه‌ی یک سطر رسیده باشد، خود تابع با ورودی شماره‌ی سطر بعدی برگردانده می‌شود، یعنی عامل به خانه‌ی بعدی که اولین ستون سطر بعدی است فرستاده می‌شود. حال تابع در اینجا بررسی می‌کند که آیا این خانه خالی و بدون مقدار است یا خیر. اگر خالی نباشد، عامل دوباره به خانه‌ی بعدی فرستاده می‌شود، اما اگر خالی باشد، حال برای این

خانه تمامی مقادیر ممکن در دامنه یعنی اعداد بین ۱ تا ۹ مورد بررسی قرار می‌گیرد که این کار به کمک یک حلقه‌ی for انجام می‌شـــود. در هر بار اجرای این حلقه، هر کدام از این اعداد به وســـیله‌ی تابع check_constraints مورد بررسی قرار گرفته می‌شوند و اگر تمامی محدودیت‌ها را ارضا کنند، بازی یک مرحله به جلو می‌رود و خانه‌ی خالی به وسـیله‌ی این عدد مقداردهی می‌شـود و سـپس صـفحه بازی نیز چاپ می‌شـــود. در ادامه تابع خودش را فراخوانی می‌کند و به خانه‌ی بعدی می‌رود و این روند تا هنگامی تکرار می‌شود که آیا با وجود این عدد، اگر بازی ادامه پیدا کند به جواب می‌رسد یا خیر، که اگر به جواب برسـد مقدار True برگردانده می‌شـود و اگر به جواب نرسـد دوباره آن خانه خالی می‌شـود تا مقدار بعدی موجود در دامنه مورد بررســـی قرار بگیرد. در انتها اگر هیچ جوابی برای پازل پیدا نشـــود، مقدار False برگردانده می‌شود. تصویر کد این تابع در شکل ۲-۱۰ نمایش داده شده است.

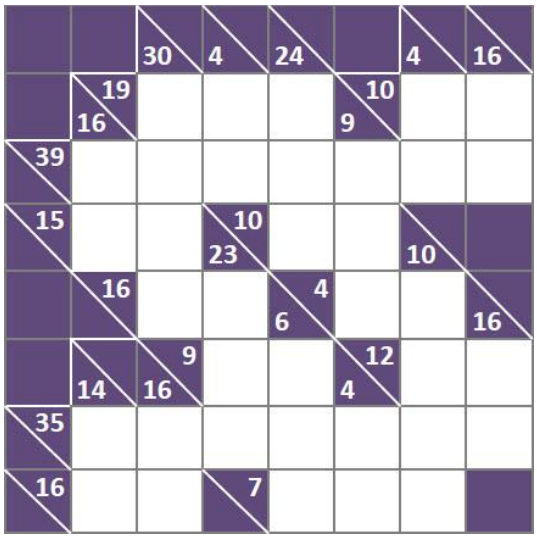
```python
def solve_kakuro_backtracking(board, row=0, col=0) -> bool:
    global level

    if row == len(board):
        return True

    if col == len(board[0]):
        return solve_kakuro_backtracking(board, row + 1, col: 0)

    if board[row][col] == "     ":
        for num in range(1, 10):
            if check_constraints(board, row, col, num):
                board[row][col] = f"  {num}  "
                print_board(board)
                print()
                level += 1
                if solve_kakuro_backtracking(board, row, col + 1):
                    return True
                board[row][col] = "     "
                print_board(board)
                print()
                level += 1

    else:
        return solve_kakuro_backtracking(board, row, col + 1)

    return False
```

شکل ۲-۱۰- کد تابع solve_kakuro_backtracking

۲-٤- نمونه اجرای بازی

در اینجا ۴ نمونه پازل Kakuro که از این لینک دریافت شده‌اند توسط عامل حل شده‌اند و نتایج آن در زیر قابل مشاهده است.

نمونه اول، یک پازل Kakuro آسان[۲] است که صفحه‌ی اولیه پازل در شکل ۲-۱۱ و نتیجه‌ی حل آن توسط عامل در شکل ۲-۱۲ نمایش داده شده است.



شکل ۲-۱۱- یک نمونه پازل Easy Kakuro

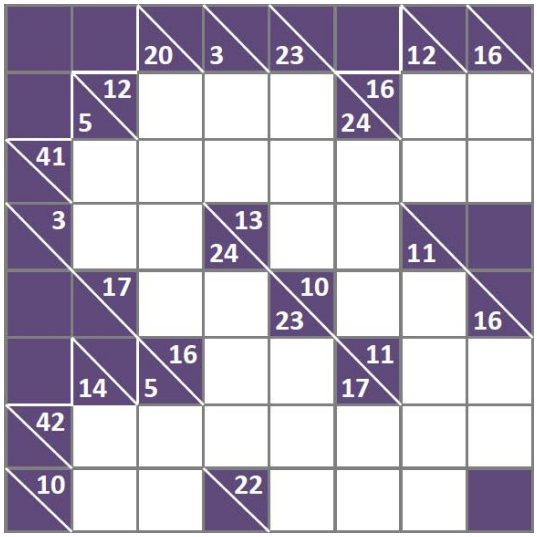
```
🔥 Yeah!!! Kakuro puzzle solved! 🔥
|   x   |   x   | 30\x  |  4\x  | 24\x  |   x   |  4\x  | 16\x  |
|   x   | 16\19 |   9   |   3   |   7   | 9\10  |   1   |   9   |
| x\39  |   9   |   6   |   1   |   8   |   5   |   3   |   7   |
| x\15  |   7   |   8   | 23\10 |   9   |   1   | 10\x  |   x   |
|   x   |  x\16 |   7   |   9   | 6\4   |   3   |   1   | 16\x  |
|   x   | 14\x  | 16\9  |   6   |   3   | 4\12  |   3   |   9   |
| x\35  |   5   |   9   |   8   |   1   |   3   |   2   |   7   |
| x\16  |   9   |   7   | x\7   |   2   |   1   |   4   |   x   |
Execution Time: 0:00:04.253899
Number of Levels: 28534
```

شکل ۲-۱۲- حل نمونه پازل Easy Kakuro توسط عامل ساده

[۲] Easy

همان‌طور که شکل ۲-۱۲ دیده می‌شود، عامل ساده توانسته است که پازل را به درستی در مدت حدودی ۴.۲۵ ثانیه و ۲۸۵۳۴ مرحله حل کند که زمان آن مطلوب ولی تعداد مراحل آن نامطلوب است.

نمونه دوم، یک پازل Kakuro متوسط[۳] است که صفحه‌ی اولیه پازل در شکل ۲-۱۳ و نتیجه‌ی حل آن توسط عامل در شکل ۲-۱۴ نمایش داده شده است.



شکل ۲-۱۳- یک نمونه پازل Medium Kakuro

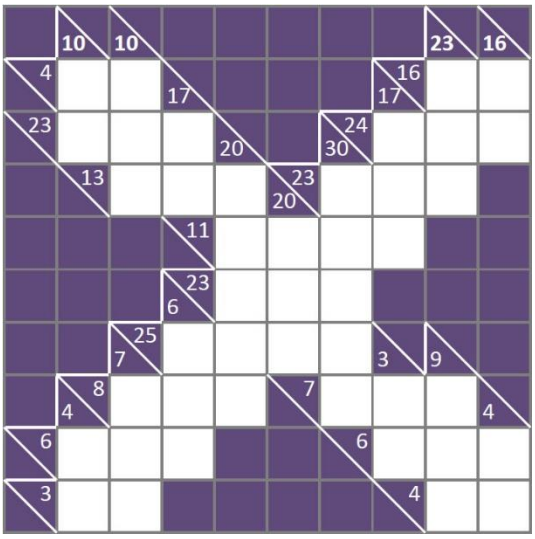
```
🔥 Yeah!!! Kakuro puzzle solved! 🔥
|   x    |   x    | 20\x   |  3\x   | 23\x   |   x    | 12\x   | 16\x   |
|   x    | 5\12   |   3    |   1    |   8    | 24\16  |   7    |   9    |
| x\41   |   4    |   6    |   2    |   9    |   8    |   5    |   7    |
| x\3    |   1    |   2    | 24\13  |   6    |   7    | 11\x   |   x    |
|   x    | x\17   |   9    |   8    | 23\10  |   9    |   1    | 16\x   |
|   x    | 14\x   |  5\16  |   7    |   9    | 17\11  |   2    |   9    |
| x\42   |   5    |   4    |   9    |   6    |   8    |   3    |   7    |
| x\10   |   9    |   1    | x\22   |   8    |   9    |   5    |   x    |
Execution Time: 0:00:02.571741
Number of Levels: 17248
```

شکل ۲-۱۴- حل نمونه پازل Medium Kakuro توسط عامل ساده

[۳] Medium

همان‌طور که شکل ۲-۱۴ دیده می‌شود، عامل ساده توانسته است که پازل را به درستی در مدت حدودی ۲.۵۷ ثانیه و ۱۷۲۴۸ مرحله حل کند که زمان آن مطلوب ولی تعداد مراحل آن نامطلوب است.

نمونه سوم، یک پازل Kakuro سخت[۴] است که صفحه‌ی اولیه پازل در شکل ۲-۱۵ و نتیجه‌ی حل آن توسط عامل در شکل ۲-۱۶ نمایش داده شده است.



شکل ۲-۱۵- یک نمونه پازل Hard Kakuro

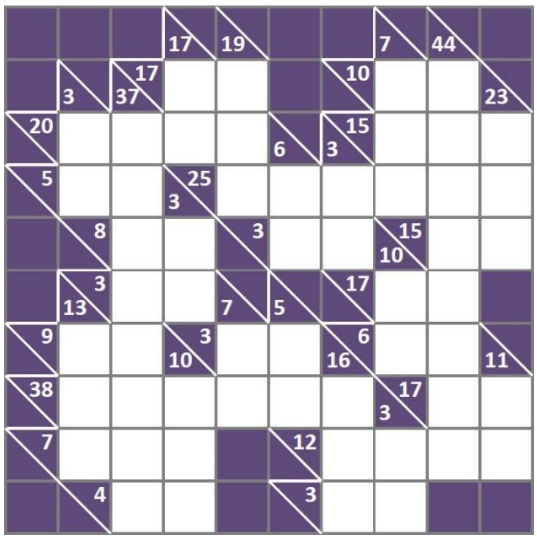
```
🔥 Yeah!!! Kakuro puzzle solved! 🔥
|   x   | 10\x  | 10\x  |   x   |   x   |   x   |   x   |   x   | 23\x  | 16\x  |
|  x\4  |   1   |   3   | 17\x  |   x   |   x   |   x   | 17\16 |   9   |   7   |
| x\23  |   9   |   6   |   8   | 20\x  |   x   | 30\24 |   7   |   8   |   9   |
|   x   | x\13  |   1   |   9   |   3   | 20\23 |   8   |   9   |   6   |   x   |
|   x   |   x   |   x   | x\11  |   2   |   3   |   5   |   1   |   x   |   x   |
|   x   |   x   |   x   | 6\23  |   6   |   8   |   9   |   x   |   x   |   x   |
|   x   |   x   | 7\25  |   1   |   8   |   9   |   7   | 3\x   | 9\x   |   x   |
|   x   | 4\8   |   4   |   3   |   1   | x\7   |   1   |   2   |   4   | 4\x   |
|  x\6  |   3   |   1   |   2   |   x   |   x   | x\6   |   1   |   2   |   3   |
|  x\3  |   1   |   2   |   x   |   x   |   x   |   x   | x\4   |   3   |   1   |
Execution Time: 0:00:00.177024
Number of Levels: 1049
```

شکل ۲-۱۶- حل نمونه پازل Hard Kakuro توسط عامل ساده

[۴] Hard

همان‌طور که شکل ۲-۱۶ دیده می‌شود، عامل ساده توانسته است که پازل را به درستی در مدت حدودی ۰.۱۷ ثانیه و ۱۰۴۹ مرحله حل کند که زمان آن بسیار مطلوب و تعداد مراحل آن نسبتا مطلوب است اما می‌تواند از این هم بهتر شود.

نمونه چهارم، یک پازل Kakuro بسیار سخت[۵] است که صفحه‌ی اولیه پازل در شکل ۲-۱۷ و نتیجه‌ی حل آن توسط عامل در شکل ۲-۱۸ نمایش داده شده است.



شکل ۲-۱۷- یک نمونه پازل Expert Kakuro

```
🔥 Yeah!!! Kakuro puzzle solved! 🔥
|   x   |   x   |   x   | 17\x  | 19\x  |   x   |   x   |  7\x  | 44\x  |   x   |
|   x   |  3\x  | 37\17 |   9   |   8   |   x   |  x\10 |   4   |   6   | 23\x  |
| x\20  |   1   |   9   |   8   |   2   |  6\x  |  3\15 |   2   |   4   |   9   |
| x\5   |   2   |   3   | 3\25  |   9   |   4   |   2   |   1   |   3   |   6   |
|   x   |  x\8  |   6   |   2   |  x\3  |   2   |   1   | 10\15 |   7   |   8   |
|   x   | 13\3  |   2   |   1   |  7\x  |  5\x  |  x\17 |   9   |   8   |   x   |
| x\9   |   4   |   5   | 10\3  |   1   |   2   | 16\6  |   1   |   5   | 11\x  |
| x\38  |   8   |   7   |   5   |   6   |   3   |   9   | 3\17  |   9   |   8   |
| x\7   |   1   |   4   |   2   |   x   |  x\12 |   6   |   1   |   2   |   3   |
|   x   |  x\4  |   1   |   3   |   x   |  x\3  |   1   |   2   |   x   |   x   |
Execution Time: 0:01:54.031296
Number of Levels: 559514
```

شکل ۲-۱۸- حل نمونه پازل Expert Kakuro توسط عامل ساده

[۵] Expert

همان‌طور که شکل ۲-۱۸ دیده می‌شود، عامل ساده توانسته است که پازل را به درستی در مدت حدودی ۱.۵۴ دقیقه و ۵۵۹۵۱۴ مرحله حل کند که زمان آن نامطلوب و تعداد مراحل آن بسیار نامطلوب است و عامل برای حل این پازل به زمان و حافظه زیادی نیاز دارد.

فصل سوم

بهبود روش پس‌گرد

بهبود روش پس‌گرد

در این فصل به بررسی کد عامل هوشمندی که در جهت بهبود عامل ساده طراحی کرده‌ایم می‌پردازیم. بعضی از توابع این عامل هوشمند، مانند تابع‌های find_horizontal_cells، find_vertical_cells، find_row_clue، find_col_clue، print_board و check_constraints با توابع عامل ساده یکسان است و بعضی از توابع نیز تغییر داده شده‌اند. بهبود این عامل و الگوریتم backtracking به وسیله‌ی بعضی از روش‌های ارضای محدودیت انجام شده است.

۳-۱- توابع کمکی

علاوه بر بعضی از توابع کمکی پیاده‌سازی شده برای عامل ساده، توابع کمکی مخصوصی برای عامل هوشمند طراحی شده است که در ادامه آن‌ها را معرفی می‌کنیم.

۳-۱-۱- تابع find_combinations_permutations

این تابع دو پارامتر number و index را دریافت می‌کند که number عدد دلخواهی است. کاری که این تابع انجام می‌دهد این است که تمامی حالت‌هایی که عدد number به صورت جمع index عدد ۱ تا ۹ غیر تکراری نوشته می‌شود را پیدا می‌کند و جایگشت‌های این حالات را نیز محاسبه می‌کند و سپس تمامی این حالات را به صورت یک لیست خروجی می‌دهد. برای مثال اگر در هنگام فراخوانی این تابع، مقدار number را برابر با ۱۵ و مقدار index را برابر با ۲ قرار دهیم، خروجی تابع به صورت زیر است:

[(٦, ٩), (٧, ٨), (٨, ٧), (٩, ٦)]

تابع این کار را به کمک تابع permutations از کتابخانه itertools انجام می‌دهد. تصویر کد این تابع در شکل ۳-۱ آورده شده است.

```python
def find_combinations_permutations(number: int, index: int) -> list:
    result = []
    if number <= 45:
        for perm in permutations(range(1, 10), index):
            if sum(perm) == number:
                result.append(perm)
    return result
```

شکل ۳-۱- تصویر کد تابع find_combinations_permutations

۳-۱-۲- تابع find_common_numbers

این تابع که تصویر کد آن در شکل ۳-۲ آورده شده است، دو لیست را به عنوان ورودی دریافت می‌کند، سپس بر اساس اولین عدد از هر عضو دو تابع، اشتراکشان را پیدا می‌کند و این اعداد را در یک لیست ذخیره و سپس از بزرگ به کوچک مرتب می‌کند. دلیل این مرتب سازی بهبود عملکرد عامل است که در ادامه بیشتر آن را بررسی خواهیم کرد.

```python
def find_common_numbers(list1: list, list2: list) -> list:
    set1 = set([item[0] for item in list1])
    set2 = set([item[0] for item in list2])

    common_numbers = list(set1.intersection(set2))
    common_numbers.sort(reverse=True)
    return common_numbers
```

شکل ۳-۲- تصویر کد تابع find_common_numbers

۳-۱-۳- تابع sort_cells_by_constraint

این تابع خانه‌های سفید در وضعیت اولیه صفحه بازی را بر اساس اینکه چقدر در محدودیت قرار گرفته‌اند، از بیشترین محدودیت تا کمترین محدودیت مرتب‌سازی می‌کند. در واقع این تابع تمام خانه‌های سفید در اول بازی که خالی هستند را پیدا می‌کند، برای هر کدام راهنماهای سطری و ستونی و طول سطر و ستون قرار گرفته در آن را محاسبه می‌کند. سپس به کمک دو تابع معرفی شده اخیر، بررسی می‌کند که هر کدام از راهنماهای هر خانه به چند حالت مجموع چند عدد (به اندازه‌ی سطر یا ستون نظیرشان) نوشته شوند، سپس این حالت‌ها را برای هر دو راهنمای یک خانه اشتراک گرفته و اندازه‌ی این لیست اشتراک گیری شده که آن را common_numbers در نظر گرفته می‌شود و همراه با طول سطر و ستون هر خانه‌ی سفید در یک لیست ذخیره می‌شود و این لیست در انتهای تابع بر اساس این دو شاخص مرتب‌سازی می‌شود و شماره سطر و ستون خانه‌های سفید مرتب شده به صورت یک لیست به عنوان خروجی برگردانده می‌شود.

```python
def sort_cells_by_constraint(board: list) -> list:
    white_cells = []

    for row in range(len(board)):
        for col in range(len(board[0])):
            if board[row][col] == "     ":
                row_clue = find_row_clue(board, row, col)
                col_clue = find_col_clue(board, row, col)
                row_length = len(find_horizontal_cells(board, row, col))
                col_length = len(find_vertical_cells(board, row, col))
                combin_perm_row_clue = find_combinations_permutations(row_clue, row_length)
                combin_perm_col_clue = find_combinations_permutations(col_clue, col_length)
                common_numbers = find_common_numbers(combin_perm_row_clue, combin_perm_col_clue)

                white_cells.append([(len(common_numbers), (row_length, col_length)), (row, col)])

    sorted_white_cells = [item[1] for item in sorted(white_cells)]

    return sorted_white_cells
```

شکل ۳-۳- تصویر کد تابع sort_cells_by_constraint

۳-۱-٤- تابع play_kakuro_csp

این تابع که تصویر کد آن در شکل ۳-۴ نیز دیده می‌شود، عملکردی مشابه تابع play_kakuro_backtracking دارد، با این تفاوت که در این تابع، ابتدا خانه‌های سفید پازل ورودی بر اساس محدودیتشان توسط تابع sort_cells_by_constraint مرتب و در لیست white_cells ذخیره می‌شوند که سپس این لیست به عنوان یکی از ورودی‌های تابع اصلی عامل هوشمند بازی، به این تابع داده می‌شود.

```python
def play_kakuro_csp(board: list):
    start_time = datetime.now()
    sorted_white_cells = sort_cells_by_constraint(board)

    print_board(board)
    print()

    if solve_kakuro_csp(board, sorted_white_cells):
        print("🔥 Yeah!!! Kakuro puzzle solved! 🔥")
        print_board(board)
    else:
        print("☹️ Oh No! I couldn't find a solution. ☹️")

    end_time = datetime.now()
    execution_time = end_time - start_time

    print(f"Execution Time: {execution_time}")
    print(f"Number of Levels: {level}")
```

شکل ۳-۴- تصویر کد تابع play_kakuro_csp

۳-۲- تابع solve_kakuro_csp

این تابع که تابع اصلی و پایه عامل هوشمند است، نسخه‌ی بهبود یافته تابع solve_kakuro_backtracking در عامل ساده است و در واقع الگوریتم اصلی این تابع همان backtracking می‌باشد که بهبود داده شده است. در این تابع به جای آنکه خانه‌های خالی به ترتیب موقعیتشان در صفحه بازی پیدا شوند و عامل هر بار به خانه‌ی بعدی در صفحه برود، ترتیب رفتن عامل به خانه‌های سفید بر اساس میزان محدودیتشان است که قبل از اجرای این تابع، توسط تابع sort_cells_by_constraint مرتب شده‌اند. در واقع عامل ابتدا به خانه‌های سفید خالی‌ای می‌رود که دارای محدودیت بیشتری هستند و در نتیجه دامنه‌ی مقادیرشان کمتر و فضای درخت جستجو کمتر می‌شود و عامل مراحل کمتری را برای رسیدن به جواب طی می‌کند. همچنین حلقه for اصلی این تابع به جای اینکه هر بار برای هر خانه‌ی خالی، مقدار ۱ تا ۹ را جایگذاری کند، این دامنه را محدودتر کرده و لیست common_numbers که نوع تعریف آن را در تابع sort_cells_by_constraint نیز بررسی کرده بودیم را به عنوان دامنه قرار می‌دهد. در نتیجه‌ی این محدود کردن دامنه، عامل هوشمند نسبت به

عامل ساده مقادیر کمتری را برای هر متغیر مورد ارزیابی و بررسی قرار می‌دهد و در نتیجه حافظه و زمان کمتری را نسبت به عامل ساده مصرف می‌کند.

تصویر کد این تابع در شکل ۳-۵ نشان داده شده است.

```python
def solve_kakuro_csp(board: list, white_cells: list, index=0) -> bool:
    global level

    if index == len(white_cells):
        return True

    row, col = white_cells[index]

    row_clue = find_row_clue(board, row, col)
    col_clue = find_col_clue(board, row, col)
    row_length = len(find_horizontal_cells(board, row, col))
    col_length = len(find_vertical_cells(board, row, col))
    combin_perm_row_clue = find_combinations_permutations(row_clue, row_length)
    combin_perm_col_clue = find_combinations_permutations(col_clue, col_length)
    common_numbers = find_common_numbers(combin_perm_row_clue, combin_perm_col_clue)

    for num in common_numbers:
        if check_constraints(board, row, col, num):
            board[row][col] = f"  {num}  "
            print_board(board)
            print()
            level += 1
            if solve_kakuro_csp(board, white_cells, index + 1):
                return True
            board[row][col] = "     "
            print_board(board)
            print()
            level += 1

    return False
```

شکل ۳-۵- تصویر کد تابع solve_kakuro_csp

۳-۳- نمونه اجرای بازی

در بخش ۲-۴ برای عامل ساده نمونه‌های از پازل Kakuro آوردیم و دیدیم که نتایج حل آن‌ها توسط این عامل چگونه بود. حال همین نمونه‌ها را به عامل هوشمند نیز دادیم و نتایج آن را در ادامه بررسی می‌کنیم.

در شکل ۳-۶ نتیجه حل پازل نمونه آسان آورده شده در شکل ۲-۱۱ نشان شده است.

```
🔥 Yeah!!! Kakuro puzzle solved! 🔥
|   x   |   x   | 30\x  |  4\x  | 24\x  |   x   |  4\x  | 16\x  |
|   x   | 16\19 |   9   |   3   |   7   | 9\10  |   1   |   9   |
| x\39  |   9   |   6   |   1   |   8   |   5   |   3   |   7   |
| x\15  |   7   |   8   | 23\10 |   9   |   1   | 10\x  |   x   |
|   x   | x\16  |   7   |   9   | 6\4   |   3   |   1   | 16\x  |
|   x   | 14\x  | 16\9  |   6   |   3   | 4\12  |   3   |   9   |
| x\35  |   5   |   9   |   8   |   1   |   3   |   2   |   7   |
| x\16  |   9   |   7   | x\7   |   2   |   1   |   4   |   x   |
Execution Time: 0:00:01.767936
Number of Levels: 362
```

شکل ۳-۶- حل نمونه پازل Easy Kakuro توسط عامل هوشمند

همان‌طور که در شکل ۳-۶ دیده می‌شود، عامل هوشمند این پازل را به درستی در زمان حدودی ۱.۷۶ ثانیه و در ۳۶۲ مرحله حل کرده است که زمان حل آن از زمان حل این پازل توسط عامل ساده کمتر و تعداد مراحل آن نیز بسیار کمتر است. در نتیجه برای این پازل الگوریتم عامل هوشمند بهینه‌تر عمل کرده است.

در شکل ۳-۷ نتیجه حل پازل نمونه متوسط آورده شده در شکل ۲-۱۳ نشان شده است.

```
🔥 Yeah!!! Kakuro puzzle solved! 🔥
|   x   |   x   | 20\x  |  3\x  | 23\x  |   x   | 12\x  | 16\x  |
|   x   | 5\12  |   3   |   1   |   8   | 24\16 |   7   |   9   |
| x\41  |   4   |   6   |   2   |   9   |   8   |   5   |   7   |
| x\3   |   1   |   2   | 24\13 |   6   |   7   | 11\x  |   x   |
|   x   | x\17  |   9   |   8   | 23\10 |   9   |   1   | 16\x  |
|   x   | 14\x  | 5\16  |   7   |   9   | 17\11 |   2   |   9   |
| x\42  |   5   |   4   |   9   |   6   |   8   |   3   |   7   |
| x\10  |   9   |   1   | x\22  |   8   |   9   |   5   |   x   |
Execution Time: 0:00:04.070300
Number of Levels: 1024
```

شکل ۳-۷- حل نمونه پازل Medium Kakuro توسط عامل هوشمند

همان‌طور که در شکل ۳-۷ دیده می‌شود، عامل هوشمند این پازل را به درستی در زمان حدودی ۴.۰۶ ثانیه و در ۱۰۲۴ مرحله حل کرده است که زمان حل آن از زمان حل این پازل توسط عامل ساده با اختلاف بسیار کمی بیشتر ولی تعداد مراحل آن نیز بسیار کمتر است. در نتیجه برای این پازل نیز الگوریتم عامل هوشمند بهینه‌تر عمل کرده است.

در شکل ۳-۸ نتیجه حل پازل نمونه سخت آورده شده در شکل ۲-۱۵ نشان شده است.

```
🔥 Yeah!!! Kakuro puzzle solved! 🔥
 |   x   | 10\x  | 10\x  |   x   |   x   |   x   |   x   |   x   | 23\x  | 16\x  |
 |  x\4  |   1   |   3   | 17\x  |   x   |   x   |   x   | 17\16 |   9   |   7   |
 | x\23  |   9   |   6   |   8   | 20\x  |   x   | 30\24 |   7   |   8   |   9   |
 |   x   | x\13  |   1   |   9   |   3   | 20\23 |   8   |   9   |   6   |   x   |
 |   x   |   x   |   x   | x\11  |   2   |   3   |   5   |   1   |   x   |   x   |
 |   x   |   x   |   x   | 6\23  |   6   |   8   |   9   |   x   |   x   |   x   |
 |   x   |   x   | 7\25  |   1   |   8   |   9   |   7   | 3\x   | 9\x   |   x   |
 |   x   | 4\8   |   4   |   3   |   1   | x\7   |   1   |   2   |   4   | 4\x   |
 |  x\6  |   3   |   1   |   2   |   x   |   x   |  x\6  |   1   |   2   |   3   |
 |  x\3  |   1   |   2   |   x   |   x   |   x   |   x   |  x\4  |   3   |   1   |
Execution Time: 0:00:00.813806
Number of Levels: 1485
```

شکل ۳-۸- حل نمونه پازل Hard Kakuro توسط عامل هوشمند

همان‌طور که در شکل ۳-۸ دیده می‌شود، عامل هوشمند این پازل را به درستی در زمان حدودی ۰.۸۱ثانیه و در ۱۴۸۵ مرحله حل کرده است که زمان حل آن از زمان حل این پازل توسط عامل ساده کمتر و تعداد مراحل آن تنها کمی بیشتر است. در نتیجه به صورت کلی برای این پازل نیز الگوریتم عامل هوشمند بهینه‌تر عمل کرده است.

در شکل ۳-۹ نتیجه حل پازل نمونه بسیار سخت آورده شده در شکل ۲-۱۷ نشان شده است.

```
🔥 Yeah!!! Kakuro puzzle solved! 🔥
 |   x   |   x   |   x   | 17\x  | 19\x  |   x   |   x   |  7\x  | 44\x  |   x   |
 |   x   |  3\x  | 37\17 |   9   |   8   |   x   |  x\10 |   4   |   6   | 23\x  |
 | x\20  |   1   |   9   |   8   |   2   | 6\x   |  3\15 |   2   |   4   |   9   |
 |  x\5  |   2   |   3   | 3\25  |   9   |   4   |   2   |   1   |   3   |   6   |
 |   x   |  x\8  |   6   |   2   |  x\3  |   2   |   1   | 10\15 |   7   |   8   |
 |   x   | 13\3  |   2   |   1   | 7\x   | 5\x   |  x\17 |   9   |   8   |   x   |
 |  x\9  |   4   |   5   | 10\3  |   1   |   2   | 16\6  |   1   |   5   | 11\x  |
 | x\38  |   8   |   7   |   5   |   6   |   3   |   9   |  3\17 |   9   |   8   |
 |  x\7  |   1   |   4   |   2   |   x   |  x\12 |   6   |   1   |   2   |   3   |
 |   x   |  x\4  |   1   |   3   |   x   |  x\3  |   1   |   2   |   x   |   x   |
Execution Time: 0:00:06.880934
Number of Levels: 418
```

شکل ۳-۹- حل نمونه پازل Expert Kakuro توسط عامل هوشمند

همان‌طور که در شکل ۳-۹ دیده می‌شود، عامل هوشمند این پازل را به درستی در زمان حدودی ۶.۸۸ ثانیه و در ۴۱۸ مرحله حل کرده است که زمان حل آن از زمان و تعداد مراحل حل این پازل توسط عامل

ساده بسیار کمتر است و تفاوت بسیار چشمگیری دارد. در نتیجه به صورت کلی برای این پازل نیز الگوریتم عامل هوشمند بهینه‌تر عمل کرده است.

با توجه به نمونه‌های آورده شده، دیدیم که الگوریتم بهبود یافته پس‌گرد نسبت به الگوریتم پس‌گرد ساده، عملکرد بسیار بهتری دارد و در حل پازل Kakuro بهینه‌تر عمل می‌کند.

فصل چهارم

لینک کد بازی Kakuro

لینک کد بازی Kakuro

همان طور که گفته شد، این بازی در محیط pycharm برنامه‌نویسی شده است، و برای به اشتراک گذاشتن با دیگران، کد حل این بازی توسط عامل ساده و عامل هوشمند در Google Colab نیز بارگزاری شده و لینک آن‌ها در زیر قابل دسترسی است:

لینک کد حل بازی Kakuro توسط عامل ساده (روش پس‌گرد) در Google Colab

لینک کد حل بازی Kakuro توسط عامل هوشمند (بهبود روش پس‌گرد) در Google Colab

فصل پنجم

جمع‌بندی و نتیجه‌گیری

جمع‌بندی و نتیجه‌گیری

در این مقاله به بررسی و تحلیل کد بازی Kakuro پرداختیم و حل کننده این بازی را توسط دو روش پس‌گرد و بهبود پس‌گرد پیاده‌سازی کردیم. دیدیم که اگر از روش‌های مختلف CSP مانند سازگاری کمان یا MRV یا... در بهبود Backtracking استفاده کنیم، دامنه‌ی جستجو توسط عامل به طرز چشمگیری کاهش می‌یابد و در نتیجه پازل در زمان کمتر و تعداد مراحل کمتری حل می‌شود و بهینه‌تر عمل می‌کند و حافظه‌ی کمتری نیز مصرف می‌شود. هدف از این تمرین نیز رسیدن به این نتیجه بود که کاهش دامنه توسط روش‌های مختلف CSP باعث بهبود عملکرد عامل هوشمند می‌شود.

منابع

سهراب جلوه‌گر، "کتاب الکترونیکی هوش مصنوعی"، ویرایش ۲۳، سال ۹۸

"Wikipedia," [Online]. Available: https://en.wikipedia.org/wiki/Kakuro.

"Geeks," [Online]. Available: https://www.geeksforgeeks.org/constraint-satisfaction-problems-csp-in-artificial-intelligence/.