Problemy spełniania ograniczeń – Sztuczna inteligencja i inżynieria wiedzy

Monika Galińska

Wstęp

Zadanie polega na zaimplementowaniu CSP dla dwóch rodzajów łamigłówek: Binary i Futoshiki. Każda posiada inny zestaw zmiennych, dziedzin i ograniczeń. Przy rozwiązywaniu ich przyjęto dwa różne podejścia. Łamigłówkę Binary zaimplementowano na dwa sposoby. W pierwszym podejściu zmienną określono jako jeden wiersz siatki (liczba zmiennych dla siatki NxN to N), natomiast w drugim typie rozwiązania oraz dla łamigłówki Futoshiki jako zmienną przyjmujemy każde pole w siatce (liczba zmiennych dla siatki NxN to N*N = N). Przyjęto tak różne podejścia dla problemu binary, by mieć możliwość porównania zachowania algorytmu w sytuacjach, kiedy mamy dużo zmiennych, ale mało dziedzin oraz kiedy mamy mało zmiennych, ale wiele dziedzin.

Baza CSP oparta jest na książce Davida Kopec „Classic Computer Science Problems in Python”. W rozdziale trzecim na przykładach wyjaśnione jest definiowanie podstawowej wersji algorytmu CSP. Baza ta została zaadaptowana do potrzeb naszego zadania. Dodano między innymi możliwość wyszukiwania wszystkich rozwiązań, w późniejszych etapach całość zostanie poszerzona o wybór heurystyk poprawiających działanie algorytmu.

Algorytm CSP:

```python
# 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]]] = {}
        self.solutions = []
        self.nodes_visited = 0
        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.")
```

Screenshot 1 Klasa CSP

Metoda sprawdzająca poprawność wartości przypisanej zmiennej, poprzez konfrontację wszystkich ograniczeń dotyczących danej zmiennej:

```python
# 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
```

Screenshot 2 Sprawdzenie poprawności przypisanej do zmiennej wartości

Implementacja algorytmu przeszukiwania z nawrotami:

```python
def backtracking_search(self, assignment=None, to_first_solution=True) -> dict[V, D] | None:
    # assignment is complete if every variable is assigned (our base case)
    if assignment is None:
        assignment = {}
    if len(assignment) == len(self.variables):
        self.solutions.append((assignment, self.nodes_visited))
        return

    # get all variables in the CSP but not in the assignment
    unassigned: List[V] = [v for v in self.variables if v not in assignment]

    # get the every possible domain value of the first unassigned variable
    first: V = unassigned[0]
    for value in self.domains[first]:
        self.nodes_visited += 1
        local_assignment = assignment.copy()
        local_assignment[first] = value
        # if we're still consistent, we recurse (continue)
        if self.consistent(first, local_assignment):
            self.backtracking_search(local_assignment, to_first_solution)
            # if we didn't find the result, we will end up backtracking
            if to_first_solution and self.solutions:
                return
    return
```

Screenshot 3 Implementacja algorytmu przeszukiwania z nawrotami

Wszystkie ograniczenia zdefiniowane dla naszych problemów rozszerzają klasę:

```python
# Base class for all constraints
class Constraint(Generic[V, D], ABC):
    # The variables that the constraint is between
    def __init__(self, variables: List[V]) -> None:
        self.variables = variables

    # Must be overridden by subclasses
    @abstractmethod
    def satisfied(self, assignment: Dict[V, D]) -> bool:
        ...
```

Screenshot 4 Klasa Constraint

Klasa abstrakcyjna reprezentująca problem, do którego stosowany jest algorytm CSP:

```python
# Base class for all problems
class Problem(ABC):
    # Solutions for problem
    def __init__(self) -> None:
        self.solutions = {}

    # Must be overridden by subclasses
    @abstractmethod
    def solve(self, n: int, grid: Grid, to_first_solution: bool) -> None:
        ...
```

Screenshot 5 Klasa reprezentująca problem

Łamigłówka Binary – zmienna jako wiersz

Dla łamigłówki Binary jako zmienne przyjęto wiersze siatki rozwiązania. W związku z tym najwięcej czasu poświęcane jest na wygenerowanie dziedzin dla każdego z wierszy. Początkowo generowane są wszystkie możliwe kombinacje, a następnie usuwane są opcje łamiące, którekolwiek z ograniczeń.

```python
def generate_domains(grid: Grid) -> dict[int, list[str]]:
    domains = {}
    for i in range(len(grid)):
        domains[i] = generate_possible(grid[i])
    return domains


def generate_possible(row: List[str]) -> list[str]:
    base_option = ['0'] * int(len(row)/2) + ['1'] * int(len(row)/2)
    possible_rows = list(set(list(itertools.permutations(base_option))))
    for i in range(len(row)):
        if row[i] != 'x':
            possible_rows = [x for x in possible_rows if x[i] == row[i]]

    res = possible_rows.copy()

    for pos_row in possible_rows:
        for i in range(2, len(pos_row)):
            if pos_row[i] == pos_row[i-1] == pos_row[i-2]:
                if pos_row in res:
                    res.remove(pos_row)
    return res
```

Screenshot 6 Generowanie dziedzin dla Binary

Ograniczenia podzielone są na trzy główne rodzaje kontrolujące unikalność wierszy i kolumn, brak więc niż dwóch tych samych cyfr koło siebie oraz tę samą liczbę 0 i 1 w każdej z kolumn (dla wierszy jest to zapewnione przy generowaniu dziedzin).

Ograniczenie: Każdy wiersz i każda kolumna są unikalne

Unikalność dla wierszy sprawdzana jest każdorazowo po przypisaniu nowej zmiennej. Dla kolumn sprawdzenie następuje dopiero po przypisaniu wszystkich zmiennych, gdyż wcześniej jest to niemożliwe do określenia.

```python
class UniqueConstraint(Constraint[int, str]):
    def __init__(self, rows: List[int]) -> None:
        super().__init__(rows)
        self.rows: List[int] = rows

    def satisfied(self, assignment: Dict[int, List[str]]) -> bool:
        # check if every row is unique
        rows = []
        for row_idx in assignment:
            rows.append(assignment[row_idx])
        if len(set(rows)) != len(rows):
            return False

        # check if every column is unique
        if len(assignment) == len(assignment[0]):
            numpy_array = np.array(rows)
            transpose = numpy_array.T
            columns = transpose.tolist()
            columns = [tuple(elem) for elem in columns]
            if len(set(columns)) != len(columns):
                return False
        return True
```

Screenshot 7 Unique Constraint for Binary

Ograniczenie: Liczba zer jest równa liczbie jedynek w każdym wierszu i kolumnie

Warunek ten sprawdzany jest po przypisaniu wszystkich zmiennych, ponieważ dopiero wtedy mamy porównanie dla pełnych kolumn.

```python
class ZerosAndOnesConstraint(Constraint[int, str]):
    def __init__(self, rows: List[int]) -> None:
        super().__init__(rows)
        self.rows: List[int] = rows

    def satisfied(self, assignment: Dict[int, List[str]]) -> bool:
        # check if number of zeros equals number of ones
        if len(assignment) == len(assignment[0]):
            rows = []
            zero_count = [0] * len(assignment[0])
            one_count = [0] * len(assignment[0])
            for row_idx in assignment:
                rows.append(assignment[row_idx])
                for i in range(len(assignment[row_idx])):
                    if assignment[row_idx][i] == '0':
                        zero_count[i] += 1
                    else:
                        one_count[i] += 1
            if not all_equal(zero_count) or not all_equal(one_count):
                return False
        return True
```

Screenshot 8 Ograniczenie liczby zer i jedynek dla Binary

Ograniczenie: W żadnej kolumnie i żadnym wierszu nie może pojawić się sekwencja trzech zer pod rząd lub trzech jedynek pod rząd

Warunek ten rozbity jest na trzy podproblemy, ponieważ po przypisaniu każdej zmiennej sprawdzane są dla każdej pozycji dwa poprzedzające wiersze, dwa następne wiersze oraz jeden poprzedzający i jeden następny wiersz. Wiersze brzegowe nie mogą mieć przypisanego sprawdzenia wszystkich warunków.

```python
class BackwardConstraint(Constraint[int, str]):
    def __init__(self, rows: List[int]) -> None:
        super().__init__(rows)
        self.rows: List[int] = rows

    def satisfied(self, assignment: Dict[int, List[str]]) -> bool:
        # check if no column has more than two identical numbers next to each other
        for row_idx in assignment:
            if row_idx-1 in assignment and row_idx-2 in assignment:
                for i in range(len(assignment[row_idx])):
                    if assignment[row_idx][i] == assignment[row_idx-1][i] == assignment[row_idx-2][i]:
                        return False
        return True
```

Screenshot 9 Backward Constraint

```python
class ForwardConstraint(Constraint[int, str]):
    def __init__(self, rows: List[int]) -> None:
        super().__init__(rows)
        self.rows: List[int] = rows

    def satisfied(self, assignment: Dict[int, List[str]]) -> bool:
        # print(assignment)
        # check if no column has more than two identical numbers next to each other
        for row_idx in assignment:
            if row_idx+1 in assignment and row_idx+2 in assignment:
                for i in range(len(assignment[row_idx])):
                    if assignment[row_idx][i] == assignment[row_idx+1][i] == assignment[row_idx+2][i]:
                        return False
        return True
```

Screenshot 10 Forward Constraint

```python
class MiddleConstraint(Constraint[int, str]):
    def __init__(self, rows: List[int]) -> None:
        super().__init__(rows)
        self.rows: List[int] = rows

    def satisfied(self, assignment: Dict[int, List[str]]) -> bool:
        # check if no column has more than two identical numbers next to each other
        for row_idx in assignment:
            if row_idx-1 in assignment and row_idx+1 in assignment:
                for i in range(len(assignment[row_idx])):
                    if assignment[row_idx][i] == assignment[row_idx-1][i] == assignment[row_idx+1][i]:
                        return False
        return True
```

Screenshot 11 Middle Constraint

Łamigłówka Binary – zmienna jako pole

W podejściu tym jedno pole siatki odpowiada jednej zmiennej. Dziedzina może dla każdej zmiennej przyjąć dwie wartości: 0 lub 1. Generowanie domen jest więc wielokrotnie szybsze od podejścia pierwszego.

```python
def generate_domains(n: int) -> dict[(int, int), list[int]]:
    domains = {}
    for i in range(n):
        for j in range(n):
            domains[(i, j)] = [0, 1]
    return domains
```

Screenshot 12 Binary v_2 generowanie domen

Ograniczenia podzielone są analogicznie jak w pierwszym podejściu na trzy główne rodzaje kontrolujące unikalność wierszy i kolumn, brak więc niż dwóch tych samych cyfr koło siebie oraz tę samą liczbę 0 i 1 w każdym z wierszy i kolumn.

Ograniczenie: Każdy wiersz i każda kolumna są unikalne

Unikalność dla wierszy sprawdzana jest po przypisaniu wszystkich zmiennych, osobno dla wierszy i kolumn.

```python
class ColumnConstraint(Constraint[tuple[int, int], int]):
    def __init__(self, fields: List[tuple[int, int]], n: int) -> None:
        super().__init__(fields)
        self.fields: List[tuple[int, int]] = fields
        self.n = n
        self.fields.sort()

    def satisfied(self, assignment: Dict[tuple[int, int], List[int]]) -> bool:
        # check if every column is unique
        lines = []
        if len(assignment) == len(self.fields):
            for i in range(self.n):
                column = []
                for j in range(self.n):
                    column.append(assignment[(i, j)])
                lines.append(column)

            if not all_unique(lines):
                return False
        return True
```

Screenshot 13 Unique Columns for Binary

```python
class RowConstraint(Constraint[tuple[int, int], int]):
    def __init__(self, fields: List[tuple[int, int]], n: int) -> None:
        super().__init__(fields)
        self.fields: List[tuple[int, int]] = fields
        self.n = n
        self.fields.sort()

    def satisfied(self, assignment: Dict[tuple[int, int], List[int]]) -> bool:
        # check if every row is unique
        lines = []
        if len(assignment) == len(self.fields):
            for i in range(self.n):
                row = []
                for j in range(self.n):
                    row.append(assignment[(i, j)])
                lines.append(row)

            if not all_unique(lines):
                return False
        return True
```

Screenshot 14 Unique Rows for Binary

Ograniczenie: Liczba zer jest równa liczbie jedynek w każdym wierszu i kolumnie

Warunek ten sprawdzany jest dla każdego wiersza i kolumny. Po przypisaniu wartości jednej ze zmiennych, następuje zweryfikowanie ograniczenia dla kolumny i wiersza, w którym znajduje się dane pole.

```python
class ZerosEqualsOnesConstraint(Constraint[tuple[int, int], int]):
    def __init__(self, line: List[tuple[int, int]]) -> None:
        super().__init__(line)
        self.line: List[tuple[int, int]] = line

    def satisfied(self, assignment: Dict[tuple[int, int], List[int]]) -> bool:
        # check if number of zeros equals number of ones
        if all(elem in assignment for elem in self.line):
            zero_count, one_count = 0, 0
            for t in self.line:
                if assignment[t] == 0:
                    zero_count += 1
                else:
                    one_count += 1
            if zero_count != one_count:
                return False
        return True
```

Screenshot 15 Ograniczenie liczby zer i jedynek dla Binary

Ograniczenie: W żadnej kolumnie i żadnym wierszu nie może pojawić się sekwencja trzech zer pod rząd lub trzech jedynek pod rząd

Ograniczenie to przyjmuje trzy sąsiadujące pola, a następnie sprawdza, jeśli to możliwe, czy przypisane im liczby są równe.

```python
class BinaryConstraint(Constraint[tuple[int, int], int]):
    def __init__(self, left: tuple[int, int], middle: tuple[int, int], right: tuple[int, int]) -> None:
        super().__init__([left, middle, right])
        self.left: tuple[int, int] = left
        self.middle: tuple[int, int] = middle
        self.right: tuple[int, int] = right

    def satisfied(self, assignment: Dict[tuple[int, int], List[int]]) -> bool:
        # check if left == middle == right
        if self.left in assignment and self.middle in assignment and self.right in assignment:
            if assignment[self.left] == assignment[self.middle] == assignment[self.right]:
                return False
        return True
```

Screenshot 16 Ograniczenie liczby tych samych cyfr pod rząd

Do algorytmu dołączone są funkcje generujące ograniczenia poszczególnych kategorii. Nie uwzględniamy ograniczenia unikalnych wierszy i kolumn, ponieważ jest ono definiowane jedno dla wszystkich zmiennych.

```python
def define_binary_constraints(grid: Grid, csp: CSP) -> None:
    for i in range(len(grid)):
        for j in range(len(grid)-2):
            csp.add_constraint(BinaryConstraint((i, j), (i, j + 1), (i, j + 2)))
    for i in range(len(grid) - 2):
        for j in range(len(grid)):
            csp.add_constraint(BinaryConstraint((i, j), (i + 1, j), (i + 2, j)))
    return


def define__zeros_equals_ones_constraints(grid: Grid, csp: CSP) -> None:
    lines = []
    for i in range(len(grid)):
        line = []
        line2 = []
        for j in range(len(grid)):
            line.append((i, j))
            line2.append((j, i))
        lines.append(line)
        lines.append(line2)
    for line in lines:
        csp.add_constraint(ZerosEqualsOnesConstraint(line))
    return
```

Screenshot 17 Generowanie ograniczeń dla Binary v_2

Łamigłówka Futoshiki

Dla łamigłówki Fushiki jako zmienne przyjęto pola siatki rozwiązania (w siatce NxN mamy N*N zmiennych). Dziedzina każdego pola jest taka sama i wynosi (dla siatki NxN) {1, 2, …, N}. Wygenerowanie zmiennych i dziedzin jest więc mało kosztowne obliczeniowo i czasowo.

```python
def generate_domains(n: int) -> dict[(int, int), list[int]]:
    domains = {}
    for i in range(n):
        for j in range(n):
            domains[(i, j)] = list(range(1, n + 1))
    return domains
```

Screenshot 18 Generowanie dziedzin dla Futoshiki

Ograniczenia podzielone są na trzy główne grupy. Sprawdzane jest czy wiersze i kolumny zawierają niepowtarzające się zestawy liczb oraz czy spełnione są ograniczenia większości i mniejszości unikalne dla każdej łamigłówki.

Ograniczenie: W każdej kolumnie i każdym wierszu znajdują się wszystkie liczby od 1 do N

Ograniczenie to podzielone zostało na dwie części: sprawdzenie dla wierszy oraz ograniczenie dla kolumn. Działają one w analogiczny sposób i wykonywane są po przypisaniu każdej zmiennej.

```python
class RowsConstraint(Constraint[tuple[int, int], int]):
    def __init__(self, variables: List[tuple[int, int]]) -> None:
        super().__init__(variables)
        self.variables: List[tuple[int, int]] = variables

    def satisfied(self, assignment: Dict[tuple[int, int], List[int]]) -> bool:
        # print(assignment)
        used = []
        for variable in self.variables:
            if variable[0] not in used:
                used.append(variable[0])
                row_list = [assignment[x] for x in assignment if variable[0] == x[0]]
                if len(row_list) != len(set(row_list)):
                    return False
        return True
```

Screenshot 19 Sprawdzenie liczb dla wierszy

```python
class ColumnsConstraint(Constraint[tuple[int, int], int]):
    def __init__(self, variables: List[tuple[int, int]]) -> None:
        super().__init__(variables)
        self.variables: List[tuple[int, int]] = variables

    def satisfied(self, assignment: Dict[tuple[int, int], List[int]]) -> bool:
        used = []
        for variable in self.variables:
            if variable[1] not in used:
                used.append(variable[1])
                column_list = [assignment[x] for x in assignment if variable[1] == x[1]]
                if len(column_list) != len(set(column_list)):
                    return False
        return True
```

Screenshot 20 Sprawdzenie liczb dla kolumn

Ograniczenie: Zdefiniowane dla każdej łamigłówki oryginalne ograniczenia mniejszości i większości

```python
class FutoshikiConstraint(Constraint[tuple[int, int], int]):
    def __init__(self, grt_field: tuple[int, int], ls_field: tuple[int, int]) -> None:
        super().__init__([grt_field, ls_field])
        self.grt_field: tuple[int, int] = grt_field
        self.ls_field: tuple[int, int] = ls_field

    def satisfied(self, assignment: Dict[tuple[int, int], List[int]]) -> bool:
        # check if grt_field > ls_field
        if self.grt_field in assignment and self.ls_field in assignment:
            if assignment[self.grt_field] <= assignment[self.ls_field]:
                return False
        return True
```

Screenshot 21 Ograniczenia mniejszości i większości

Ograniczenia te wgrywane są z siatki danej łamigłówki.

```python
def define_futoshiki_constraints(grid: Grid, csp: CSP) -> None:
    cosntr_row, cosntr_col = [], []
    for i in range(len(grid)):
        if (i % 2) == 0:
            cosntr_row.append([])
            for j in range(len(grid[i])):
                if grid[i][j] == '>' or grid[i][j] == '<' or grid[i][j] == '-':
                    cosntr_row[int(i / 2)].append(grid[i][j])
        else:
            cosntr_col.append([])
            for j in range(len(grid[i])):
                if grid[i][j] == '>' or grid[i][j] == '<' or grid[i][j] == '-':
                    cosntr_col[int(i / 2)].append(grid[i][j])

    for i in range(len(cosntr_row)):
        for j in range(len(cosntr_row[i])):
            if cosntr_row[i][j] == '>':
                csp.add_constraint(FutoshikiConstraint((i, j), (i, j + 1)))
            if cosntr_row[i][j] == '<':
                csp.add_constraint(FutoshikiConstraint((i, j + 1), (i, j)))

    for i in range(len(cosntr_col)):
        for j in range(len(cosntr_col[i])):
            if cosntr_col[i][j] == '>':
                csp.add_constraint(FutoshikiConstraint((i, j), (i + 1, j)))
            if cosntr_col[i][j] == '<':
                csp.add_constraint(FutoshikiConstraint((i + 1, j), (i, j)))
    return
```

Screenshot 22 Definiowanie ograniczeń na podstawie siatki

Forward checking

Forward checking polega na sprawdzaniu czy wartość, którą chcemy przypisać zmiennej nie spowoduje, że którakolwiek z dziedzin nieprzypisanych zmiennych stanie się pusta. Ograniczamy także dziedziny nieprzypisanych zmiennych poprzez usuwanie z nich wartości konfliktowych z obecnym stanem grafu.

```python
def forward_checking_search(self, assignment=None, to_first_solution=True, MRV=False, LSC=False) -> dict[V, D] | None:
    if assignment is None:
        assignment = {}
    if len(assignment) == len(self.variables):
        self.solutions.append((assignment, self.nodes_visited))
        return

    unassigned: List[V] = [v for v in self.variables if v not in assignment]
    if MRV:
        first: V = self.MRV(unassigned)
    else:
        first: V = unassigned[0]

    if LSC:
        values = self.LSC(first, assignment, unassigned)
    else:
        values = self.domains[first]
```

```
for value in values:
    self.nodes_visited += 1
    local_assignment = assignment.copy()
    local_assignment[first] = value

    if self.consistent(first, local_assignment):
        emptyDomainFound = False
        for variable in self.get_unassigned_from_constraints(first, unassigned[1:]):
            wipe, new_domain = self.forward_check(variable, local_assignment)
            if wipe:
                emptyDomainFound = True
                self.domains = copy.deepcopy(self.domains_copy)
                break
            else:
                self.domains[variable] = new_domain
        if not emptyDomainFound:
            self.forward_checking_search(local_assignment, to_first_solution)
            if to_first_solution and self.solutions:
                return
return
```

Metoda forward_check sprawdza dla zmiennej czy obecne przypisanie powoduje zredukowanie dziedziny do zbioru pustego.

```
def forward_check(self, variable, assignment):
    values = copy.deepcopy(self.domains_copy[variable])
    res = []
    for value in values:
        temp_assignment = copy.deepcopy(assignment)
        temp_assignment[variable] = value
        if self.consistent(variable, temp_assignment):
            res.append(value)
    return len(res) == 0, res
```

Heurystki

Minimum Remaining Values – wybór zmiennej

Wybór zmiennej w tym przypadku polega na znalezieniu nieprzypisanej zmiennej z najmniejszą dziedziną.

```python
def MRV(self, unassigned):
    best_variable = unassigned[0]
    best_length = len(self.domains[best_variable])
    for var in unassigned[1:]:
        curr_length = len(self.domains[var])
        if curr_length < best_length:
            best_variable = var
            best_length = curr_length
    return best_variable
```

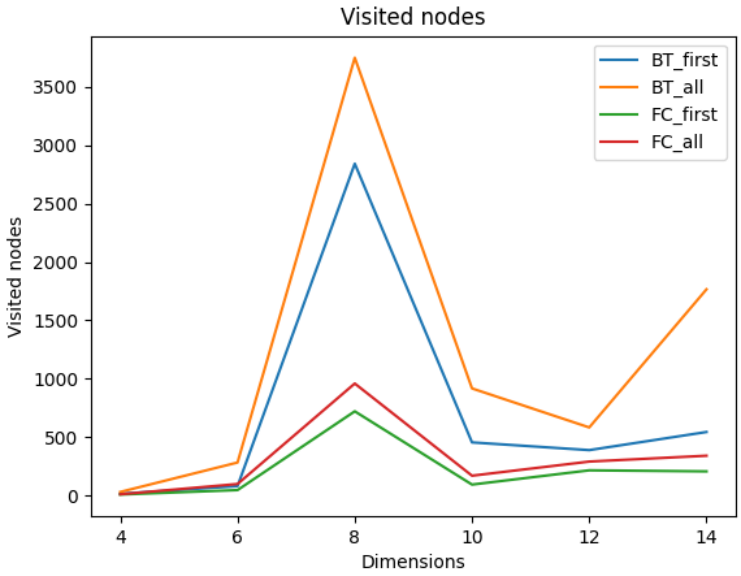
Least Constraining Value – wybór wartości

Wybór polega na znalezieniu takiej wartości, której przypisanie wyklucza najmniejszą liczbę wartości z dziedzin sąsiadujących zmiennych.

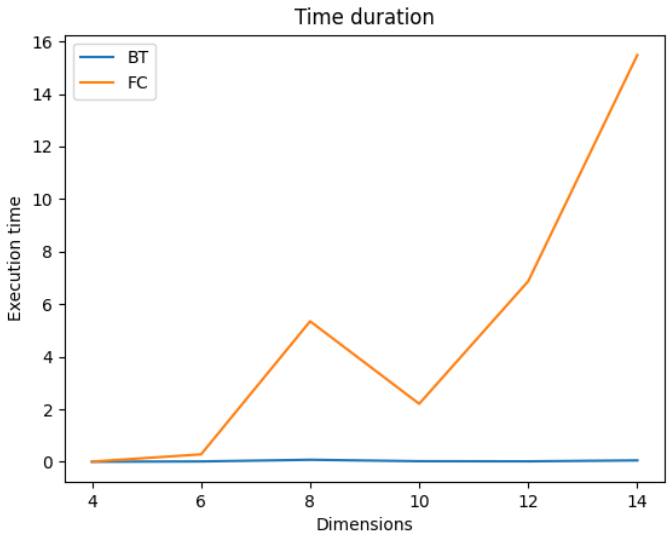
```
def LSC(self, variable, assignment, unassigned):
    ordered_domain = {}
    unassigned_const = self.get_unassigned_from_constraints(variable, unassigned)
    temp_assignment = copy.deepcopy(assignment)
    for value in self.domains[variable]:
        ordered_domain[value] = 0
        temp_assignment[variable] = value
        for var in unassigned_const:
            for dom in self.domains[var]:
                temp_assignment[var] = dom
                if not self.consistent(var, temp_assignment):
                    ordered_domain[value] += 1
    ordered = {k: v for k, v in sorted(ordered_domain.items(), key=lambda item: item[1], reverse=False)}
    return ordered.keys()
```

Binary

Porównanie BT i FC bez uwzględnienia heurystyk:

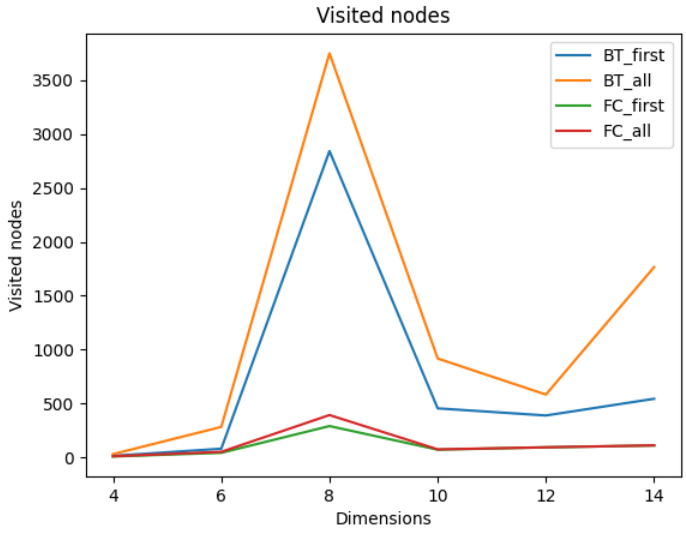


N		4	6	8	10	12	14
BT	FST	16	82	2841	456	390	545
BT	ALL	32	284	3748	918	584	1766
FC	FST	10	48	722	95	217	208
FC	ALL	12	100	960	171	292	342

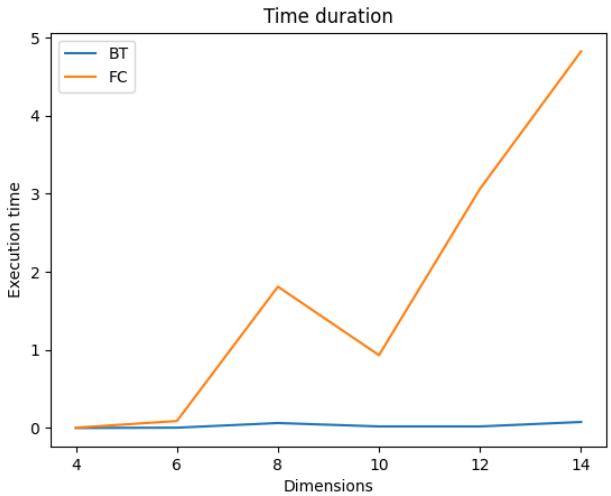


N	4	6	8	10	12	14
BT	0.002	0.014	0.077	0.023	0.018	0.054
FC	0.008	0.283	5.348	2.207	6.874	15.479

Porównanie BT i FC z zastosowaniem MRV:

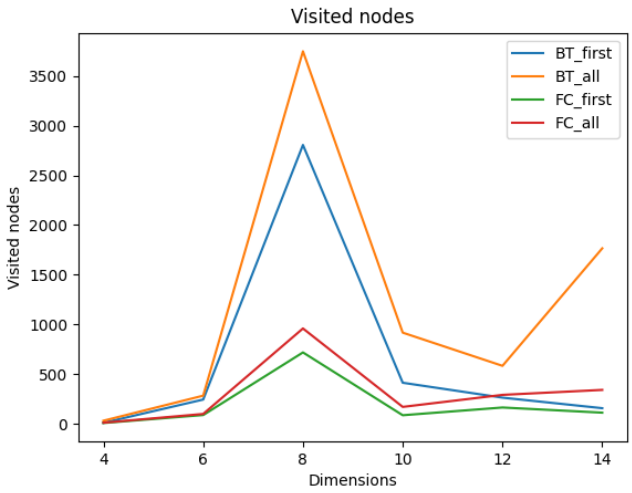


N		4	6	8	10	12	14
BT	FST	16	82	2841	456	390	545
BT	ALL	32	284	3748	918	584	1766
FC	FST	10	44	292	73	95	112
FC	ALL	12	53	394	76	95	113

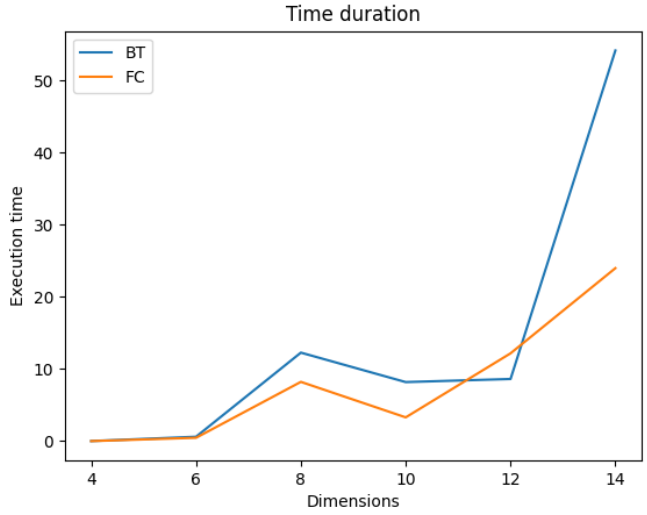


N	4	6	8	10	12	14
BT	0.0	0.004	0.063	0.02	0.02	0.076
FC	0.004	0.09	1.81	0.933	3.063	4.823

Porównanie BT i FC z zastosowaniem LCV:

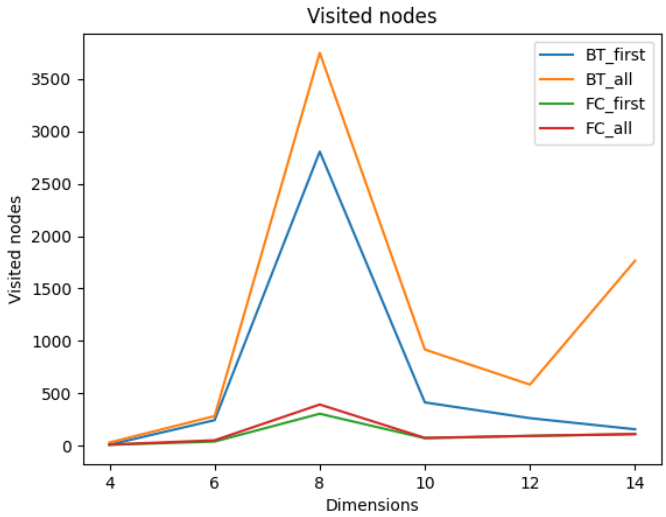


N		4	6	8	10	12	14
BT	FST	11	245	2806	414	264	158
BT	ALL	32	284	3748	918	584	1766
FC	FST	9	89	719	87	165	113
FC	ALL	12	100	960	171	292	342

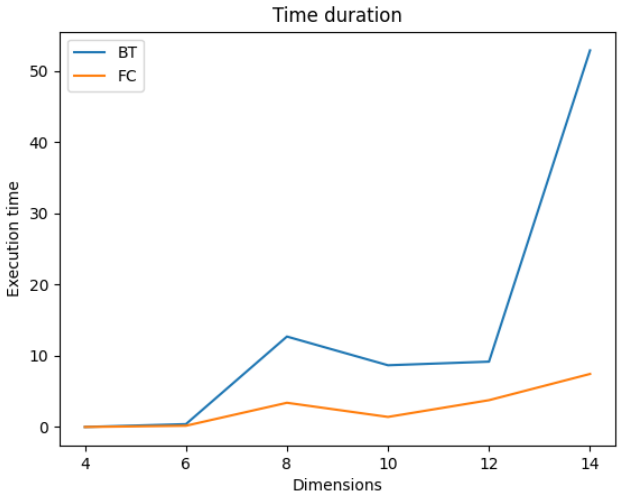


N	4	6	8	10	12	14
BT	0.011	0.61	12.264	8.183	8.613	54.136
FC	0.007	0.464	8.213	3.293	12.168	23.969

Porównanie BT i FC z zastosowaniem MRV + LCV:



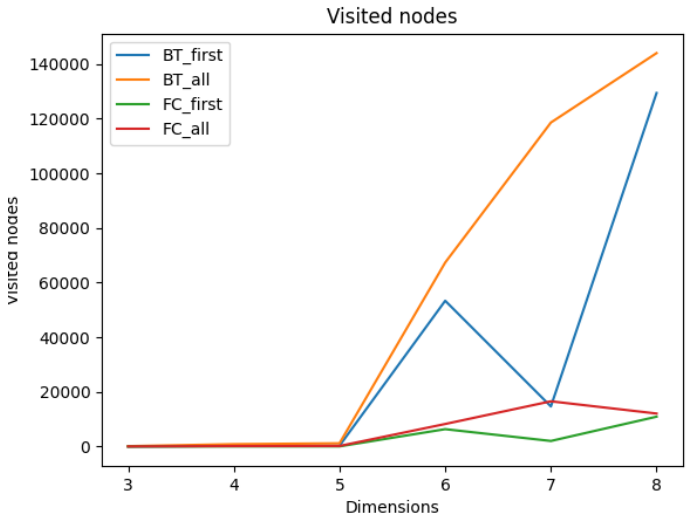
N		4	6	8	10	12	14
BT	FST	11	245	2806	414	264	158
BT	ALL	32	284	3748	918	584	1766
FC	FST	10	41	306	73	95	111
FC	ALL	12	53	394	76	95	113



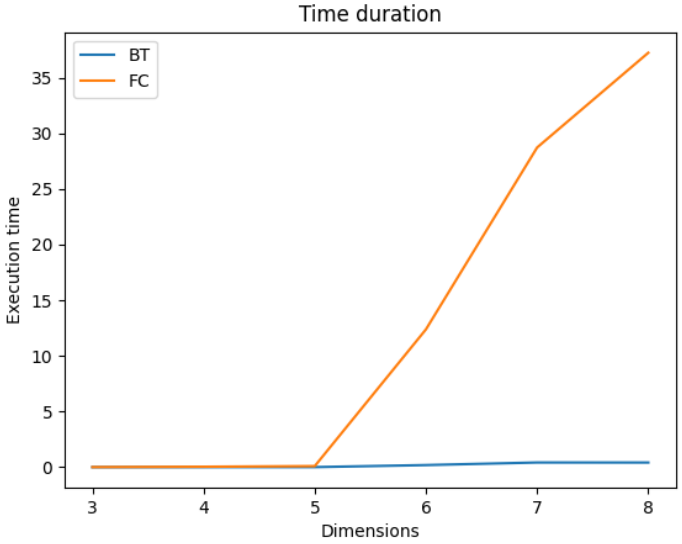
N	4	6	8	10	12	14
BT	0.011	0.409	12.706	8.676	9.181	52.882
FC	0.009	0.184	3.409	1.422	3.773	7.455

Futoshiki

Porównanie BT i FC bez zastosowania heurystyk:

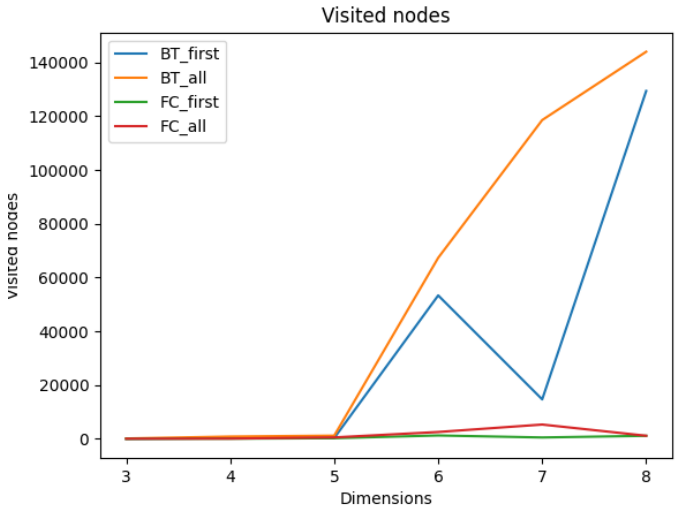


N		3	4	5	6	7	8
BT	FST	18	265	296	53375	14685	129448
BT	ALL	132	864	1190	67368	118622	144048
FC	FST	9	58	55	6361	2036	10931
FC	ALL	32	165	205	8256	16561	12079

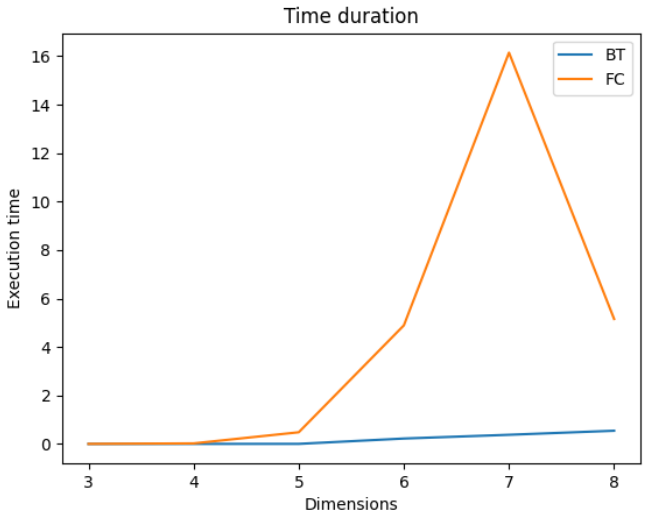


N	4	6	8	10	12	14
BT	0.0	0.002	0.003	0.187	0.419	0.415
FC	0.003	0.037	0.094	12.382	28.725	37.240

Porównanie BT i FC z zastosowaniem MRV:

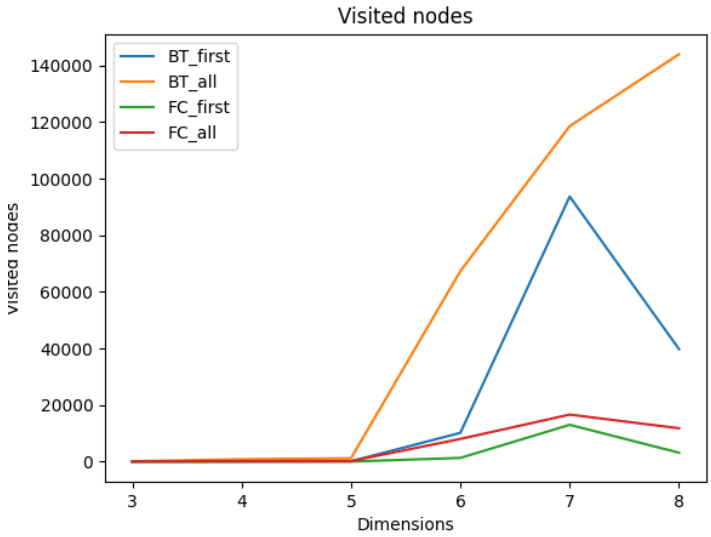


N		3	4	5	6	7	8
BT	FST	18	265	296	53375	14685	129448
BT	ALL	132	864	1190	67368	118622	144048
FC	FST	9	68	241	1252	508	1127
FC	ALL	16	85	532	2547	5318	1144

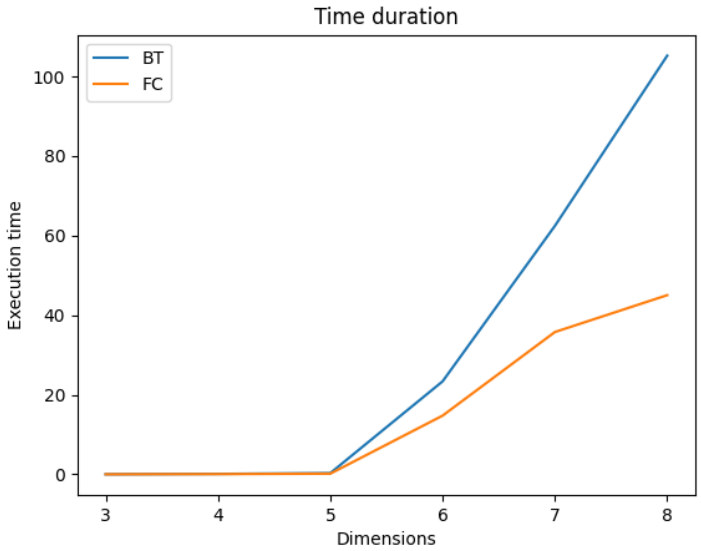


N	3	4	5	6	7	8
BT	0.001	0.003	0.004	0.224	0.379	0.545
FC	0.003	0.024	0.482	4.893	16.144	5.157

Porównanie BT i FC z zastosowaniem LCV:

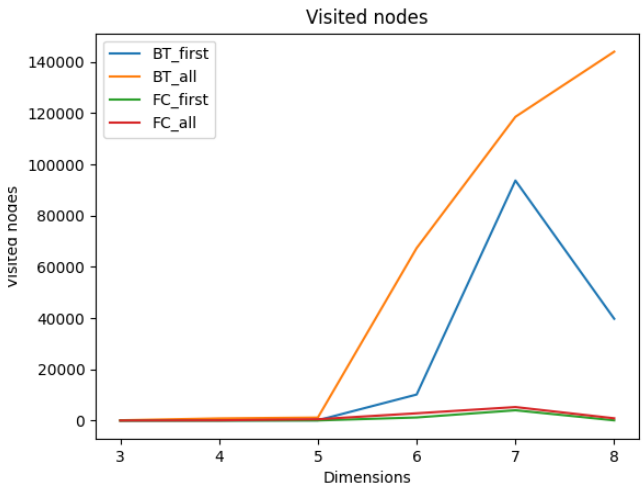


N		3	4	5	6	7	8
BT	FST	18	39	126	10179	93736	39761
BT	ALL	132	864	1190	67368	118622	144048
FC	FST	9	17	27	1317	13010	3161
FC	ALL	32	157	194	8017	16632	11804

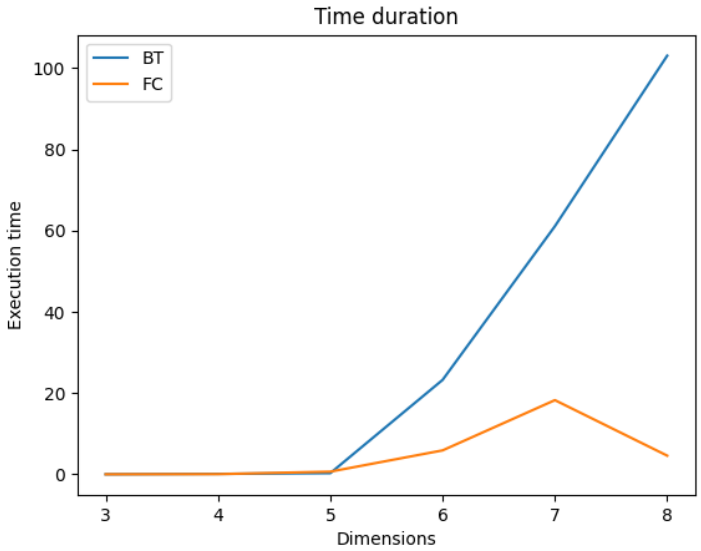


N	3	4	5	6	7	8
BT	0.008	0.099	0.297	23.360	62.438	105.205
FC	0.006	0.058	0.160	14.747	35.760	45.022

Porównanie BT i FC z zastosowaniem MRV + LCV:



N		3	4	5	6	7	8
BT	FST	18	39	126	10179	93736	39761
BT	ALL	132	864	1190	67368	118622	144048
FC	FST	9	16	69	1226	4037	100
FC	ALL	36	118	532	2852	5288	901



N	3	4	5	6	7	8
BT	0.009	0.107	0.279	23.250	61.089	103.077
FC	0.006	0.045	0.652	5.891	18.263	4.590

Porównanie czasu działania algorytmu w zależności od doboru heurystyk

Base	4	6	8	10	12	14
BT	0.002	0.014	0.077	0.023	0.018	0.054
FC	0.008	0.283	5.348	2.207	6.874	15.479
MRV	4	6	8	10	12	14
BT	0.0	0.004	0.063	0.02	0.02	0.076
FC	0.004	0.09	1.81	0.933	3.063	4.823
LCV	4	6	8	10	12	14
BT	0.011	0.61	12.264	8.183	8.613	54.136
FC	0.007	0.464	8.213	3.293	12.168	23.969
MRV+LCV	4	6	8	10	12	14
BT	0.011	0.409	12.706	8.676	9.181	52.882
FC	0.009	0.184	3.409	1.422	3.773	7.455

Base	4	6	8	10	12	14
BT	0.0	0.002	0.003	0.187	0.419	0.415
FC	0.003	0.037	0.094	12.382	28.725	37.240
MRV	3	4	5	6	7	8
BT	0.001	0.003	0.004	0.224	0.379	0.545
FC	0.003	0.024	0.482	4.893	16.144	5.157
LCV	3	4	5	6	7	8
BT	0.008	0.099	0.297	23.360	62.438	105.205
FC	0.006	0.058	0.160	14.747	35.760	45.022
MRV+LCV	3	4	5	6	7	8
BT	0.009	0.107	0.279	23.250	61.089	103.077
FC	0.006	0.045	0.652	5.891	18.263	4.590

Dla backtracking najbardziej optymalne czasowo jest MRV oraz brak heurystyk. Dodanie LCV gwałtownie pogarsza czas działania algorytmu (z ok. 0.5s do ponad 100s dla Futoshiki 14x14), jest więc nieopłacalny przy implementacji BT. Dla FC Możemy zauważyć przeciwny trend. Samo LCV nieznacznie pogarsza osiągany czas, jednak w połączeniu z MRV otrzymujemy w części przypadków czas lepszy od algorytmu bez heurystyk. W przypadku FC korzystne jest więc zaimplementowanie LCV.

Porównanie liczby odwiedzonych węzłów w zależności od doboru heurystyk

W przypadku rozpatrywanych łamigłówek zastosowanie MRV dla BT nie wnosi żadnych zmian ze względu na identyczną liczność dziedzin wszystkich zmiennych. MRV nie zmienia kolejności zmiennych, a więc także liczba odwiedzonych węzłów pozostaje niezmienna. Dla FC dodanie MRV kilku- a nawet kilkunastokrotnie zmniejsza liczbę odwiedzonych węzłów. Jest to możliwe ze względu na zmieniającą się w trakcie działania FC liczbę wartości w dziedzinach zmiennych.

LCV w porównaniu do wersji bazowej i MRV wypada różnie. Dla siatek 4x4 i 5x5 Futorashiki daje najlepsze efekty z porównywanej trójki. Jednak w przypadku gridu 7x7 powoduje przejście przez kilkukrotnie więcej węzłów.

Przy połączeniu działania MRV z LCV ponownie otrzymujemy mieszane efekty. Dla części przypadków dają one najlepszy wynik, dla innych liczba węzłów wypada gorzej niż dla samego MRV.

Base		3	4	5	6	7	8
BT	FST	18	265	296	53375	14685	129448
BT	ALL	132	864	1190	67368	118622	144048
FC	FST	9	58	55	6361	2036	10931
FC	ALL	32	165	205	8256	16561	12079
MRV		**3**	**4**	**5**	**6**	**7**	**8**
BT	FST	18	265	296	53375	14685	129448
BT	ALL	132	864	1190	67368	118622	144048
FC	FST	9	68	241	1252	508	1127
FC	ALL	16	85	532	2547	5318	1144
LCV		**3**	**4**	**5**	**6**	**7**	**8**
BT	FST	18	39	126	10179	93736	39761
BT	ALL	132	864	1190	67368	118622	144048
FC	FST	9	17	27	1317	13010	3161
FC	ALL	32	157	194	8017	16632	11804
MRV + LCV		**3**	**4**	**5**	**6**	**7**	**8**
BT	FST	18	39	126	10179	93736	39761
BT	ALL	132	864	1190	67368	118622	144048
FC	FST	9	16	69	1226	4037	100
FC	ALL	36	118	532	2852	5288	901

Podsumowanie

Porównanie czasu działania algorytmu oraz liczby węzłów w zależności od stosowanych heurystyk pokazuje, że dla BT nieopłacalnym jest stosowanie LCV ze względu na znaczący koszt czasowy. Także dodanie MRV, mimo niskiego kosztu czasowego, w przypadku badanych łamigłówek jest zbędne, ponieważ nie zmniejsza liczby odwiedzanych węzłów. W przypadku innych problemów MRV mogłoby okazać się korzystne pod warunkiem, że implementacja zawierałaby zróżnicowaną liczność dziedzin zmiennych.

Podczas porównywania uzyskanych wyników w tabeli można zauważyć także, że dla BT nie zmienia się liczba węzłów dla poszukiwania wszystkich rozwiązań. Wynika to z faktu, że w żadnym momencie działania algorytmu nie eliminujemy wartości z dziedzin zmiennych. Algorytm musi więc sprawdzić każdorazowo taką samą liczbę węzłów, by mieć pewność znalezienia wszystkich rozwiązań. Heurystyki wpływają jedynie na kolejność tego przeszukiwania, możemy więc stosując np. LCV uzyskać pierwsze poprawne rozwiązanie znacznie szybciej niż w wersji podstawowej.

BT vs FC

Porównując działanie BT i FC okazuje się, że FC jest o wiele bardziej korzystny od BT, gdy patrzymy na ilość przeszukiwanych węzłów. Problemem może być powolny czas działania w wersji podstawowej. Problem ten rozwiązuje jednak zastosowanie odpowiednik heurystyk, które pozwalają osiągać znacznie lepszą liczbę odwiedzonych węzłów, przy zachowaniu czasu działania porównywalnego do wersji bazowej. Ostatecznie zatem FC pozwala na kilkunastokrotne zmniejszenie liczby przeszukanych węzłów, kosztem niewielkiego spowolnienia działania algorytmu.