```python
def __init__(self, courses, students, time_slots):
    """
    :param courses: list of course names (variables)
    :param students: dict mapping student_id -> list of enrolled courses
    :param time_slots: list of available time slots (e.g., [1,2,3])
    """
    self.courses : list = copy.deepcopy(courses)
    self.students : dict = students
    self.time_slots : list = time_slots
    # TODO: create variables list
    self.variables : list = copy.deepcopy(courses)
    # TODO: initialize domains for each course (variable)
    self.domains : dict = self.build_domains()

    # TODO: build conflict graph (constraints) and store in self.conflicts
    # conflicts should be a dict mapping course -> set(of conflicting courses)
    self.conflicts : dict = self.build_constraints()
```

وظیفه:

این تابع سازنده (Constructor) کلاس است و داده‌های اولیه مسئله را دریافت و آماده می‌کند.

کارهایی که انجام می‌دهد:

1. لیست درس‌ها (courses) را ذخیره می‌کند.
2. اطلاعات دانشجویان (students) را ذخیره می‌کند.
3. لیست زمان‌های امتحان (time_slots) را ذخیره می‌کند.
4. متغیرها :(variables) همان لیست درس‌هاست.
5. دامنه‌ها :(domains) برای هر درس لیستی از زمان‌های امتحان ممکن ایجاد می‌کند.
6. گراف تداخل :(conflicts) برای هر درس، لیست درس‌هایی که نمی‌توانند همزمان برگزار شوند را ذخیره می‌کند.

```python
def build_domains(self):
    domains = dict()
    for course in self.courses:
            domains[course] =  list(self.time_slots)

    return domains

def build_constraints(self):
    """
    Build and return conflict graph.
    If two courses share any student, they conflict (cannot share same timeslot).
    """
    conflict = dict()

    for course in self.courses:
        conflict[course] = set()

    for courses in self.students.values():
        for course in courses:
            for course_2 in courses:
                if course_2 != course:
                    conflict[course].add(course_2)


    # TODO: implement and return a dict mapping course -> set(conflicting_courses)
    return conflict
```

تابع build domain

دامنه (Domain) هر متغیر را می‌سازد.

دامنه یعنی لیستی از مقادیر ممکن که آن درس می‌تواند بگیرد.

کاری که انجام می‌دهد:

برای هر درس، تمام زمان‌های امتحان را در دامنه‌اش قرار می‌دهد.

تابع build constraints

ساخت گراف تداخل

این گراف مشخص می‌کند که کدام درس‌ها نمی‌توانند در یک زمان امتحان داده شوند.

```python
def is_consistent(self, assignment : dict, course, value):
    """
    Check if assigning 'value' to 'course' is consistent with current partial assignment.
    Use self.conflicts to check neighbors that already have assignments.

    :param assignment: dict course->timeslot (partial)
    :param course: candidate variable
    :param value: candidate timeslot
    :return: True if no conflict, False otherwise
    """
    # TODO: check for any neighbor in assignment that has same timeslot
    for neighbor in self.conflicts[course]:
        if neighbor in assignment and assignment[neighbor] == value:
            return False
    return True
```

تابع is_consistent:

بررسی می‌کند که آیا اختصاص دادن یک زمان امتحان به یک درس با constraints ها سازگار است یا خیر.

کاری که انجام می‌دهد:

- همه درس‌های همسایه (درس‌هایی که با این درس تداخل دارند) را بررسی می‌کند.
- اگر یکی از همسایه‌ها قبلاً همان زمان امتحان را گرفته باشد، نتیجه False است.
- در غیر اینصورت True برمی‌گرداند

```python
# ------------------------
# Plain Backtracking solver
# ------------------------
def backtracking_search(self):
    """Run plain backtracking on this CSP and return an assignment or None."""
    return self._backtrack({})

def _backtrack(self, assignment):
    # TODO: if assignment is complete return assignment
    if len(assignment) == len(self.courses):
        return assignment

    # TODO: select an unassigned variable (naive: first unassigned)
    var = ''
    for v in self.variables:
        if v not in assignment:
            var = v
            break

    if var == '': return None

    # TODO: iterate over domain values for var
    for value in self.domains[var]:
        if self.is_consistent(assignment,var,value):
            assignment[var] = value
            result = self._backtrack(assignment)

            if result is not None:
                return result

            assignment.pop(var)


    return None
```

تابع Backtracking

کاری که انجام می‌دهد:

1. اگر همه درس‌ها زمان امتحان گرفته باشند assignment کامل است.
2. انتخاب یک متغیر (درس) بدون زمان امتحان.
3. برای هر مقدار ممکن (زمان امتحان) درس:
   - بررسی سازگاری با is_consistent
   - اگر سازگار بود ← آن را اضافه کن و ادامه بده.
4. اگر هیچ زمان امتحانی مناسب پیدا نشود return None و متغیر را از assigned حذف میکند

```python
def select_unassigned_variable(self, assignment : dict , local_domains):
    MRV: choose variable with minimum remaining values (domain size).
    Degree heuristic as a tie-breaker: choose variable with most conflicts (neighbors).
    :param assignment: current partial assignment
    :param local_domains: dict course->list(current_domain_values), used by FC
    :return: selected variable name
    """
    # TODO: implement MRV + Degree tie-breaker

    MRV_set = list()
    min_domain = 9999999
    for items in local_domains:
        if items in assignment.keys():
            continue
        items_len = len(local_domains[items])
        if items_len < min_domain:
            min_domain = len(local_domains[items])
            MRV_set = list()
            MRV_set.append(items)
        elif items_len == min_domain:
            MRV_set.append(items)

    if len(MRV_set) > 1 :
        max_conf = -99999
        var = ''
        for item in MRV_set:
            count = 0
            for conflict in self.conflicts[item]:
                if conflict not in assignment.keys():
                    count +=1
            if max_conf <= count:
                max_conf = count
                var = item
        return var
    else:
        return MRV_set.pop()
```

تابع select_unassigned_variable :

انتخاب متغیر بعدی با استفاده از MRV و Degree heuristic

کاری که انجام می‌دهد:

- MRV (Minimum Remaining Values) درس با کمترین تعداد گزینه‌های ممکن را انتخاب میکند.
- Degree heuristic اگر چند درس با یک تعداد گزینه باقی‌مانده وجود داشته باشد ، درس با بیشترین تداخل را انتخاب میکند.

```python
def forward_check(self, var, value, assignment, local_domains):
    """
    Remove 'value' from domains of unassigned neighbors (in local_domains).
    Record removed pairs in a list and return (success, removed_list).
    If a neighbor domain becomes empty, return (False, removed_list).
    """
    # TODO: implement forward checking; do not modify self.domains directly.
    record_list = list()

    for neighbor in self.conflicts[var]:
        if neighbor not in assignment.keys():
            if value in local_domains[neighbor]:
                local_domains[neighbor].remove(value)
                record_list.append((neighbor,value))
                if len(local_domains[neighbor]) <= 0 :
                    return False,record_list
    return True, record_list
```

تابع forward_check

کاری که انجام می‌دهد :

اگر یک درس زمان امتحان گرفت ، این زمان را از دامنه همسایه‌های آن درس حذف میکند.

خروجی:

- True یا False همراه با لیستی از (زوج‌های درس، زمان حذف‌شده) برای بازگرداندن دامنه در صورت شکست

```python
def _backtrack_heuristic(self, assignment : dict, local_domains):
    # TODO: if assignment complete → return assignment
    if len(assignment) == len(self.courses):
        return assignment
    # TODO: select variable using select_unassigned_variable(assignment, local_domains)
    var = self.select_unassigned_variable(assignment,local_domains)
    # TODO: iterate through values in local_domains[var] (consider ordering)

    for value in local_domains[var]:
        if self.is_consistent(assignment,var,value):
            assignment[var] = value
            status,record_list = self.forward_check(var,value,assignment,local_domains)
            if not status:
                assignment.pop(var)
                for var_2,value_2 in record_list:
                        local_domains[var_2].append(value_2)
            else:
                result = self._backtrack_heuristic(assignment,local_domains)
                if result is not None:
                    return result

                assignment.pop(var)
                for var_2,value_2 in record_list:
                        local_domains[var_2].append(value_2)
    # For each value:
    #   - if consistent, assign
    #   - perform forward_check to prune neighbor domains
    #   - if forward_check OK, recurse with updated local_domains
    #   - restore domains from removed list if recursion fails
    #   - unassign variable and continue
```

این تابع یک الگوریتم Backtracking پیشرفته است که از MRV + Degree + Forward Checking استفاده می‌کند تا CSP را سریع‌تر حل کند.

کارهایی که انجام می‌دهد:

1. بررسی کامل بودن پاسخ.
2. انتخاب متغیر بعدی با استفاده از MRV + Degree
3. بررسی مقادیر ممکن.
4. بررسی Constriant ها برای مقدار انتخابی.
5. اعمال Forward Checking برای حذف مقدار داده شده از همسایه متغیر.
6. اگر Forward Checking شکست خورد ← بازگشت و برگرداندن مقادیر حذف شده به دامنه متغیر ها
7. اگر موفق بود ← ادامه تا پیدا شدن پاسخ کامل.

شماره تست کیس	BACKTRACK عادی	BACKTRACK پیشرفته
1	0.000011	0.000023
2	0.000011	0.000024
3	0.000010	0.000022
4	0.000012	0.000048
5	0.000012	0.000068
6	0.000020	0.000057
7	0.000023	0.000086
8	0.000032	0.000145

نتیجه :

========== Test Case 1 ==========

Courses: ['Math', 'Physics']

Students: {'s1': ['Math', 'Physics']}

Time slots: [1, 2]

-- Plain Backtracking (student implementation) --

Solution (plain): {'Math': 1, 'Physics': 2}

Execution Time: 0.000011 seconds

-- Backtracking + MRV + Degree + Forward Checking (student implementation) --

Solution (heuristic): {'Physics': 1, 'Math': 2}

Execution Time: 0.000023 seconds

---------- End Test Case 1 ----------

========== Test Case 2 ==========

Courses: ['A', 'B', 'C']

Students: {'s1': ['A', 'B'], 's2': ['B', 'C']}

[2 ,1] :Time slots

-- Plain Backtracking (student implementation) --

Solution (plain): {'A': 1, 'B': 2, 'C': 1}

Execution Time: 0.000011 seconds

-- Backtracking + MRV + Degree + Forward Checking (student implementation) --

Solution (heuristic): {'B': 1, 'C': 2, 'A': 2}

Execution Time: 0.000024 seconds

---------- End Test Case 2 ----------

========== Test Case 3 ==========

Courses: ['X', 'Y', 'Z']

Students: {'s1': ['X'], 's2': ['Y'], 's3': ['Z']}

[2 ,1] :Time slots

-- Plain Backtracking (student implementation) --

Solution (plain): {'X': 1, 'Y': 1, 'Z': 1}

Execution Time: 0.000010 seconds

-- Backtracking + MRV + Degree + Forward Checking (student implementation) --

Solution (heuristic): {'Z': 1, 'Y': 1, 'X': 1}

Execution Time: 0.000022 seconds

---------- End Test Case 3 ----------

========== Test Case 4 ==========

Courses: ['AI', 'ML', 'DB', 'OS']

Students: {'s1': ['AI', 'ML'], 's2': ['ML', 'DB'], 's3': ['AI', 'OS']}

[3 ,2 ,1] :Time slots

-- Plain Backtracking (student implementation) --

Solution (plain): {'AI': 1, 'ML': 2, 'DB': 1, 'OS': 2}

Execution Time: 0.000012 seconds

-- Backtracking + MRV + Degree + Forward Checking (student implementation) --

Solution (heuristic): {'ML': 1, 'AI': 2, 'OS': 1, 'DB': 2}

Execution Time: 0.000048 seconds

---------- End Test Case 4 ----------

========== Test Case 5 ==========

Courses: ['C1', 'C2', 'C3', 'C4', 'C5']

Students: {'s1': ['C1', 'C2'], 's2': ['C2', 'C3'], 's3': ['C3', 'C4'], 's4': ['C4', 'C5']}

[3 ,2 ,1] :Time slots

-- Plain Backtracking (student implementation) --

Solution (plain): {'C1': 1, 'C2': 2, 'C3': 1, 'C4': 2, 'C5': 1}

Execution Time: 0.000012 seconds

-- Backtracking + MRV + Degree + Forward Checking (student implementation) --

Solution (heuristic): {'C4': 1, 'C3': 2, 'C2': 1, 'C5': 2, 'C1': 2}

Execution Time: 0.000068 seconds

---------- End Test Case 5 ----------

========== Test Case 6 ==========

Courses: ['P', 'Q', 'R', 'S']

Students: {'a': ['P', 'Q'], 'b': ['Q', 'R'], 'c': ['R', 'S'], 'd': ['P', 'S'], 'e': ['P', 'R']}

[3 ,2 ,1] :Time slots

-- Plain Backtracking (student implementation) --

Solution (plain): {'P': 1, 'Q': 2, 'R': 3, 'S': 2}

Execution Time: 0.000020 seconds

-- Backtracking + MRV + Degree + Forward Checking (student implementation) --

Solution (heuristic): {'R': 1, 'P': 2, 'S': 3, 'Q': 3}

Execution Time: 0.000057 seconds

---------- End Test Case 6 ----------

========== Test Case 7 ==========

Courses: ['M1', 'M2', 'M3', 'M4', 'M5', 'M6']

Students: {'s1': ['M1', 'M2'], 's2': ['M2', 'M3'], 's3': ['M3', 'M4'], 's4': ['M4', 'M5'], 's5': ['M5', 'M6'], 's6': ['M1', 'M6']}

[3 ,2 ,1] :Time slots

-- Plain Backtracking (student implementation) --

Solution (plain): {'M1': 1, 'M2': 2, 'M3': 1, 'M4': 2, 'M5': 1, 'M6': 2}

Execution Time: 0.000023 seconds

-- Backtracking + MRV + Degree + Forward Checking (student implementation) --

Solution (heuristic): {'M6': 1, 'M5': 2, 'M4': 1, 'M3': 2, 'M2': 1, 'M1': 2}

Execution Time: 0.000086 seconds

---------- End Test Case 7 ----------

========== Test Case 8 ==========

Courses: ['CS1', 'CS2', 'CS3', 'CS4', 'CS5', 'CS6', 'CS7', 'CS8']

Students: {'u1': ['CS1', 'CS2'], 'u2': ['CS2', 'CS3'], 'u3': ['CS3', 'CS4'], 'u4': ['CS4', 'CS5'], 'u5': ['CS5', 'CS6'], 'u6': ['CS6', 'CS7'], 'u7': ['CS7', 'CS8'], 'u8': ['CS8', 'CS1'], 'u9': ['CS1', 'CS3'], 'u10': ['CS2', 'CS4']}

[4 ,3 ,2 ,1] :Time slots

-- Plain Backtracking (student implementation) --

Solution (plain): {'CS1': 1, 'CS2': 2, 'CS3': 3, 'CS4': 1, 'CS5': 2, 'CS6': 1, 'CS7': 2, 'CS8': 3}

Execution Time: 0.000032 seconds

-- Backtracking + MRV + Degree + Forward Checking (student implementation) --

Solution (heuristic): {'CS4': 1, 'CS3': 2, 'CS2': 3, 'CS1': 1, 'CS8': 2, 'CS7': 1, 'CS6': 2, 'CS5': 3}

Execution Time: 0.000145 seconds

---------- End Test Case 8 ----------

تحلیل :

در حالت کلی سرعت backtrack + heuristic باید بیشتر از حالت عادی باشد ولی در این جا به دلیل کوچک بودن تست کیس ها نتیجه برعکس است و backtrack عادی بهتر از حالت پیشرفته آن عمل کرده است.

دلیل :

1. Backtracking زمان اجرای کمتری برای مسائل ساده دارد چونکه اعمال هیوریستیک و چک کردن دامنه را انجام نمیدهد به همین دلیل Heuristic Backtracking (MRV + Degree + FC) برای مسائل کوچک نسبت به Backtracking کندتر است

با افزایش اندازه و پیچیدگی مسئله، مزیت الگوریتم هیوریستیک واضح تر است.