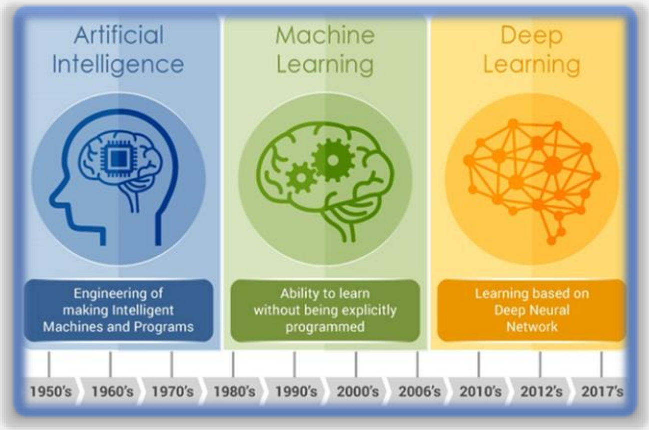
CENG 3511 Artificial Intelligence

Week 4
Constraint Satisfaction Problems


Artificial Intelligence
Machine Learning
Deep Learning
Engineering of making Intelligent Machines and Programs
Ability to learn without being explicitly programmed
Learning based on Deep Neural Network
1950's
1960's
1970's
1980's
1990's
2000's
2006's
2010's
2012's
2017's

Instructor
Bekir Taner Dinçer
Teaching Assistant
Selahattin Aksoy

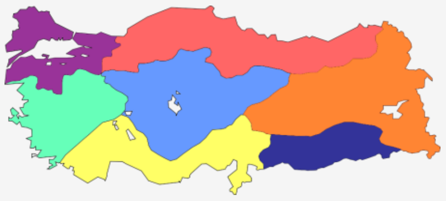
MUĞLA SITKI KOÇMAN UNIVERSITY
COMPUTER ENGINEERING
2024 – FALL

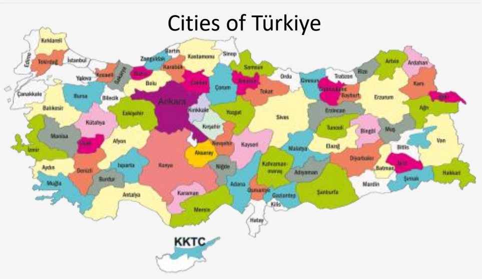
1

Constraint Satisfaction Problems - CSPs

Map Coloring Problem:
- How many different colors we need to color the regions of Türkiye without having two adjacent regions with the same color, or
- How many colors are need to color the cities of Türkiye having no two adjacent cities with the same color.

Regional Map of Türkiye





2

Constraint Satisfaction Problems - CSPs

Scheduling Problems

- **Timetabling**: Assigning courses, teachers, and classrooms to specific time slots while avoiding conflicts.

Resource Allocation Problems, e.g.,

- **Facility Location**: Determining the optimal locations for facilities (e.g., warehouses, schools) to serve a given population.

Other Problems

- **Artificial Intelligence Planning**: Generating plans to achieve goals in complex environments.
- **Natural Language Processing**: Parsing sentences and understanding their meaning.

Puzzle Problems

- **Einstein's Riddle**: Solving a logic puzzle involving five houses, five people of different nationalities, five different pets, five different cigarettes, and five different drinks.
- **Cryptograms**: Deciphering a coded message by assigning letters to symbols.

3

Constraint Satisfaction Problems - CSPs

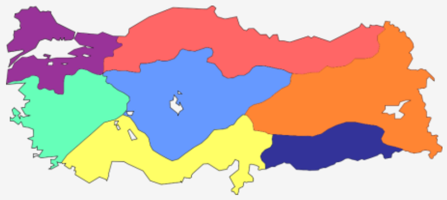
Definition

- A CSP is a problem that the solution we need is to find a valid assignment of **values** (*colors*) to **variables** (*cities*) such that all specified **constraints** (*no adjacent cities with the same color*) are satisfied, without violating any of the problem's rules.

Key Components:

- **Variables**: The entities to which values must be assigned (e.g., cities in a map coloring problem).
- **Domains**: The set of possible values that each variable can take (e.g., color options in map coloring)
- **Constraints**: Rules that restrict the ways variables can be assigned values (e.g., adjacent cities must have different colors)

Regional Map of Türkiye



Cities of Türkiye
KKTC

4

Types of Constraints

1. **Unary Constraints**
   - Constraints that apply to a single variable
2. **Binary Constraints**
   - Constraints that involve two variables and restrict the values that can be assigned to both
3. **Higher-Order Constraints**
   - Constraints that involve three or more variables simultaneously

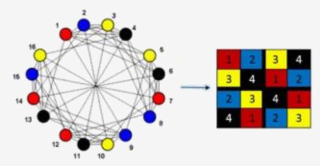
Unary Constraints: Examples

Map Coloring:

- In a map coloring problem, a unary constraint could specify that a particular country, say Country A, must be colored red.

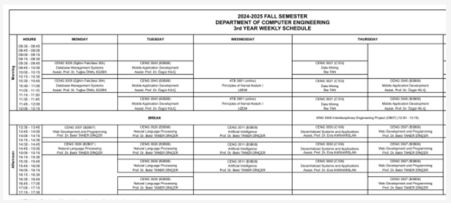
Sudoku:

- In Sudoku, a unary constraint might state that a specific cell (e.g., top-left) must be a certain value, such as 5.



Timetabling:

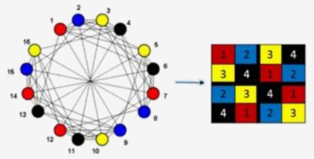
- In a timetabling problem, a unary constraint might require that a particular course (e.g., "Math 101") must be scheduled in a specific classroom (e.g., Room 204).

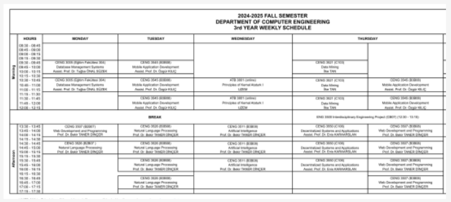


2024-2025 FALL SEMESTER
DEPARTMENT OF COMPUTER ENGINEERING
3rd YEAR WEEKLY SCHEDULE

	HOURS	MONDAY	TUESDAY	WEDNESDAY	THURSDAY	
Morning	08:30 - 08:45 08:45 - 09:00 09:00 - 09:15 09:15 - 09:30					
	09:30 - 09:45 09:45 - 10:00 10:00 - 10:15	CENG 3005 (Eğitim Fakültesi 304) Database Management Systems Assist. Prof. Dr. Tuğba ÖNAL SÜZEK	CENG 3545 (B3808) Mobile Application Development Assist. Prof. Dr. Özgür KILIÇ		CENG 3021 (C103) Data Mining İlke TAN	
	10:15 - 10:30					
	10:30 - 10:45 10:45 - 11:00 11:00 - 11:15	CENG 3005 (Eğitim Fakültesi 304) Database Management Systems Assist. Prof. Dr. Tuğba ÖNAL SÜZEK	CENG 3545 (B3808) Mobile Application Development Assist. Prof. Dr. Özgür KILIÇ	ATB 3001 (online) Principles of Kemal Atatürk I UZEM	CENG 3021 (C103) Data Mining İlke TAN	CENG 3545 (B3808) Mobile Application Development Assist. Prof. Dr. Özgür KILIÇ
	11:15 - 11:30					
	11:30 - 11:45 11:45 - 12:00 12:00 - 12:15		CENG 3545 (B3808) Mobile Application Development Assist. Prof. Dr. Özgür KILIÇ	ATB 3001 (online) Principles of Kemal Atatürk I UZEM	CENG 3021 (C103) Data Mining İlke TAN	CENG 3545 (B3808) Mobile Application Development Assist. Prof. Dr. Özgür KILIÇ
		BREAK			ENG 3000 Interdisciplinary Engineering Project (C807) (12:30 - 13:15)	
Afternoon	13:30 - 13:45 13:45 - 14:00 14:00 - 14:15	CENG 3507 (B2807) Web Development and Programming Prof. Dr. Bekir TANER DİNÇER	CENG 3526 (B3808) Natural Language Processing Prof. Dr. Bekir TANER DİNÇER	CENG 3511 (B3808) Artificial Intelligence Prof. Dr. Bekir TANER DİNÇER	CENG 3550 (C106) Decentralized Systems and Applications Assist. Prof. Dr. Enis KARAARSLAN	CENG 3507 (B3808) Web Development and Programming Prof. Dr. Bekir TANER DİNÇER
	14:15 - 14:30					
	14:30 - 14:45 14:45 - 15:00 15:00 - 15:15	CENG 3526 (B2807) Natural Language Processing Prof. Dr. Bekir TANER DİNÇER	CENG 3526 (B3808) Natural Language Processing Prof. Dr. Bekir TANER DİNÇER	CENG 3511 (B3808) Artificial Intelligence Prof. Dr. Bekir TANER DİNÇER	CENG 3550 (C106) Decentralized Systems and Applications Assist. Prof. Dr. Enis KARAARSLAN	CENG 3507 (B3808) Web Development and Programming Prof. Dr. Bekir TANER DİNÇER
	15:15 - 15:30					
	15:30 - 15:45 15:45 - 16:00 16:00 - 16:15		CENG 3526 (B3808) Natural Language Processing Prof. Dr. Bekir TANER DİNÇER	CENG 3511 (B3808) Artificial Intelligence Prof. Dr. Bekir TANER DİNÇER	CENG 3550 (C106) Decentralized Systems and Applications Assist. Prof. Dr. Enis KARAARSLAN	CENG 3507 (B3808) Web Development and Programming Prof. Dr. Bekir TANER DİNÇER
	16:15 - 16:30					
	16:30 - 16:45 16:45 - 17:00 17:00 - 17:15		CENG 3526 (B3808) Natural Language Processing Prof. Dr. Bekir TANER DİNÇER			CENG 3507 (B3808) Web Development and Programming Prof. Dr. Bekir TANER DİNÇER
	17:15 - 17:30					

Binary Constraints: Examples





2024-2025 FALL SEMESTER
DEPARTMENT OF COMPUTER ENGINEERING
3rd YEAR WEEKLY SCHEDULE

	HOURS	MONDAY	TUESDAY	WEDNESDAY	THURSDAY	
Morning	08:30 - 08:45 08:45 - 09:00 09:00 - 09:15 09:15 - 09:30					
	09:30 - 09:45 09:45 - 10:00 10:00 - 10:15 10:15 - 10:30	CENG 3005 (Eğitim Fakültesi 304) Database Management Systems Assist. Prof. Dr. Tuğba ÖNAL SÜZEK	CENG 3545 (B3B08) Mobile Application Development Assist. Prof. Dr. Özgür KILIÇ		CENG 3521 (C103) Data Mining İlke TAN	
	10:30 - 10:45 10:45 - 11:00 11:00 - 11:15 11:15 - 11:30	CENG 3005 (Eğitim Fakültesi 304) Database Management Systems Assist. Prof. Dr. Tuğba ÖNAL SÜZEK	CENG 3545 (B3B08) Mobile Application Development Assist. Prof. Dr. Özgür KILIÇ	ATB 3001 (online) Principles of Kemal Atatürk I UZEM	CENG 3521 (C103) Data Mining İlke TAN	CENG 3545 (B3B08) Mobile Application Development Assist. Prof. Dr. Özgür KILIÇ
	11:30 - 11:45 11:45 - 12:00 12:00 - 12:15		CENG 3545 (B3B08) Mobile Application Development Assist. Prof. Dr. Özgür KILIÇ	ATB 3001 (online) Principles of Kemal Atatürk I UZEM	CENG 3521 (C103) Data Mining İlke TAN	CENG 3545 (B3B08) Mobile Application Development Assist. Prof. Dr. Özgür KILIÇ
		BREAK			ENG 3500 Interdisciplinary Engineering Project (CB07) (12:30 - 13:15)	
Afternoon	13:30 - 13:45 13:45 - 14:00 14:00 - 14:15 14:15 - 14:30	CENG 3507 (B2B07) Web Development and Programming Prof. Dr. Bekir TANER DİNÇER	CENG 3526 (B3B08) Natural Language Processing Prof. Dr. Bekir TANER DİNÇER	CENG 3511 (B3B08) Artificial Intelligence Prof. Dr. Bekir TANER DİNÇER	CENG 3550 (C106) Decentralized Systems and Applications Assist. Prof. Dr. Enis KARAARSLAN	CENG 3507 (B3B08) Web Development and Programming Prof. Dr. Bekir TANER DİNÇER
	14:30 - 14:45 14:45 - 15:00 15:00 - 15:15 15:15 - 15:30	CENG 3526 (B2B07) Natural Language Processing Prof. Dr. Bekir TANER DİNÇER	CENG 3526 (B3B08) Natural Language Processing Prof. Dr. Bekir TANER DİNÇER	CENG 3511 (B3B08) Artificial Intelligence Prof. Dr. Bekir TANER DİNÇER	CENG 3550 (C106) Decentralized Systems and Applications Assist. Prof. Dr. Enis KARAARSLAN	CENG 3507 (B3B08) Web Development and Programming Prof. Dr. Bekir TANER DİNÇER
	15:30 - 15:45 15:45 - 16:00 16:00 - 16:15 16:15 - 16:30		CENG 3526 (B3B08) Natural Language Processing Prof. Dr. Bekir TANER DİNÇER	CENG 3511 (B3B08) Artificial Intelligence Prof. Dr. Bekir TANER DİNÇER	CENG 3550 (C106) Decentralized Systems and Applications Assist. Prof. Dr. Enis KARAARSLAN	CENG 3507 (B3B08) Web Development and Programming Prof. Dr. Bekir TANER DİNÇER
	16:30 - 16:45 16:45 - 17:00 17:00 - 17:15 17:15 - 17:30		CENG 3526 (B3B08) Natural Language Processing Prof. Dr. Bekir TANER DİNÇER			CENG 3507 (B3B08) Web Development and Programming Prof. Dr. Bekir TANER DİNÇER

Map Coloring:

- A binary constraint could be between two neighboring countries, ensuring that Country A ≠ Country B (they cannot share the same color).

Sudoku:

- A binary constraint could require that two cells in the same row or column have different numbers, such as Cell A ≠ Cell B.

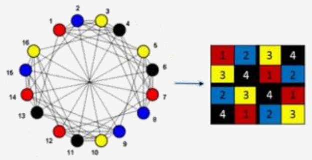
Timetabling:

- In timetabling, a binary constraint could ensure that two courses taught by the same teacher (e.g., "Math 101" and "Physics 101") do not overlap in time.

Binary Constraints: Examples

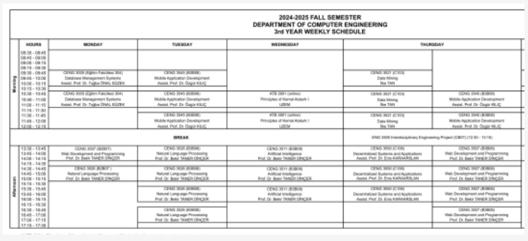
Sudoku:

- A higher-order constraint is applied to the 3x3 subgrids in Sudoku, where all cells within the subgrid must contain different values.



Timetabling:

- In timetabling, a higher-order constraint might ensure that a course, a teacher, and a classroom are all available at the same time without conflict. For instance, "Math 101" should be assigned to "Professor A" in "Room 204" at a time when all are free, without overlapping with other courses or teachers.



Working Example

CENG3511-AI-Lab3-ContrainSatisfactionProblems.ipynb

Problem statement:

Assign colors to regions such that no two adjacent regions have the same color.

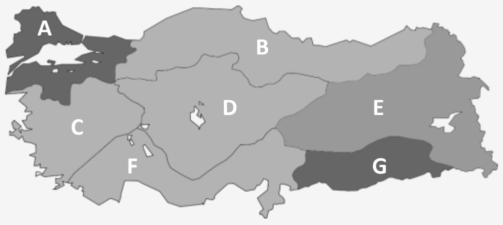
Alternative statement:

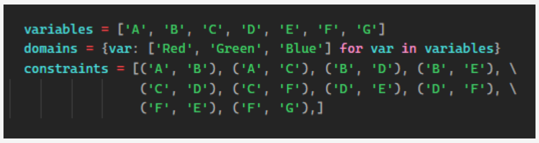
How many colors do we need at minimum to color the regions with different colors?

Variables : Regions (A, B, C, D, E, F, G)

Domains : {Red, Green, Blue}

Constraints : Adjacent regions cannot share the same color.





```
variables = ['A', 'B', 'C', 'D', 'E', 'F', 'G']
domains = {var: ['Red', 'Green', 'Blue'] for var in variables}
constraints = [('A', 'B'), ('A', 'C'), ('B', 'D'), ('B', 'E'), \
               ('C', 'D'), ('C', 'F'), ('D', 'E'), ('D', 'F'), \
               ('F', 'E'), ('F', 'G'),]
```

domains = {'A': ['Red', 'Green', 'Blue'], 'B': ['Red', 'Green', 'Blue'], … }

Here, (‘A’, ‘B’) means region A and B cannot have the same color: that is, they are adjacent.

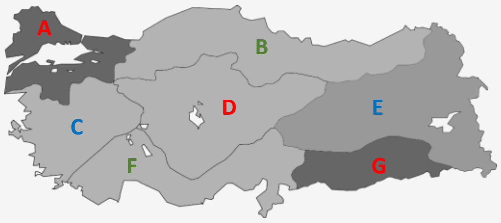
Basic CSP Solving Techniques

Backtracking and Forward Checking

Backtracking

Pseudocode

1. Attempt to assign a color to each variable.
2. After each assignment, check if it is valid (consistent with the constraints).
3. If valid, move on to the next variable (recursive).
4. If not valid, backtrack and try a different color.
5. Continue until either a solution is found, or all possibilities are exhausted.



Solution: {'A': 'Red', 'B': 'Green', 'C': 'Blue', 'D': 'Red', 'E': 'Blue', 'F': 'Green', 'G': 'Red'}

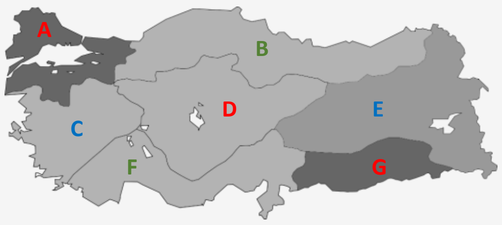
Backtracking

```python
def backtracking(assignment):
    # Base case: if all variables are assigned, return
    if len(assignment) == len(variables):
        return assignment

    # Select an unassigned variable
    var = [v for v in variables if v not in assignment][0]
    for value in domains[var]:
        if is_valid(assignment, var, value):
            assignment[var] = value
            result = backtracking(assignment)
            if result:
                return result
            assignment.pop(var)  # Backtrack

    return None

solution = backtracking({})
```

Recursion

```python
variables = ['A', 'B', 'C', 'D', 'E', 'F', 'G']
domains = {var: ['Red', 'Green', 'Blue'] for var in variables}
constraints = [('A', 'B'), ('A', 'C'), ('B', 'D'), ('B', 'E'), \
               ('C', 'D'), ('C', 'F'), ('D', 'E'), ('D', 'F'), \
               ('F', 'E'), ('F', 'G'),]
```

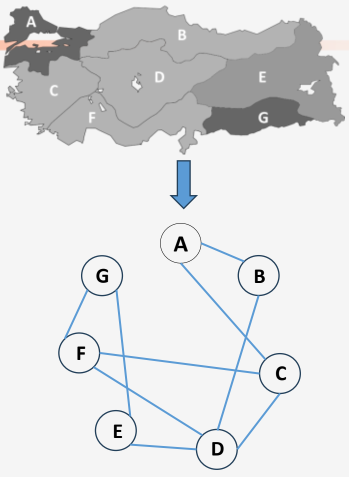
domains = {'A': ['Red', 'Green', 'Blue'], 'B': ['Red', 'Green', 'Blue'], ... }



Solution: {'A': 'Red', 'B': 'Green', 'C': 'Blue', 'D': 'Red', 'E': 'Blue', 'F': 'Green', 'G': 'Red'}

Graph Coloring

Map Coloring problem
can be modelled as a
Graph Coloring Problem



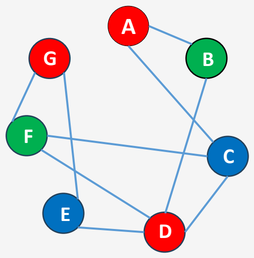
Key Points:

- **Vertices**: represent regions or cities or countries.
- **Edges**: represent adjacency.

Graph Coloring

Map Coloring problem
can be modelled as a
Graph Coloring Problem

Solution: {'A': 'Red', 'B': 'Green', 'C': 'Blue', 'D': 'Red', 'E': 'Blue', 'F': 'Green', 'G': 'Red'}



Key Points:

- **Vertices**: represent regions or cities or countries.
- **Edges**: represent adjacency.

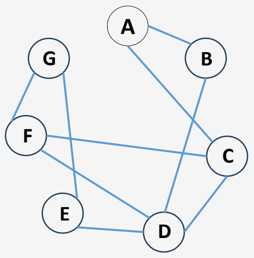
Backtracking: Trace

```
def backtracking(assignment):
    # Base case: if all variables are assigned, return
    if len(assignment) == len(variables):
        return assignment

    # Select an unassigned variable
    var = [v for v in variables if v not in assignment][0]
    for value in domains[var]:
        if is_valid(assignment, var, value):
            assignment[var] = value
            result = backtracking(assignment)
            if result:
                return result
            assignment.pop(var)  # Backtrack

    return None

solution = backtracking({})
```

variables = ['A', 'B', 'C', 'D', 'E', 'F', 'G']

domains = {'A': ['Red', 'Green', 'Blue'], 'B': ['Red', 'Green', 'Blue'], … }

assignment = {}



15

Backtracking: Trace

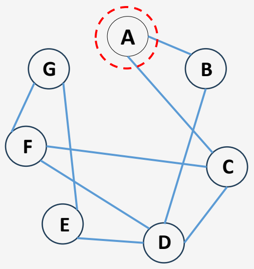
```
def backtracking(assignment):
    # Base case: if all variables are assigned, return
    if len(assignment) == len(variables):
        return assignment

    # Select an unassigned variable
    var = [v for v in variables if v not in assignment][0]
    for value in domains[var]:
        if is_valid(assignment, var, value):
            assignment[var] = value
            result = backtracking(assignment)
            if result:
                return result
            assignment.pop(var)  # Backtrack

    return None

solution = backtracking({})
```

variables = ['A', 'B', 'C', 'D', 'E', 'F', 'G']

domains = {'A': ['Red', 'Green', 'Blue'], 'B': ['Red', 'Green', 'Blue'], … }

assignment = {}

var = A



16

Backtracking: Trace

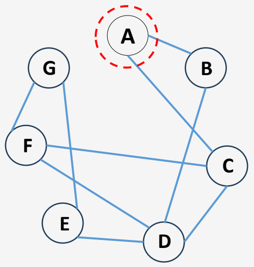
```
def backtracking(assignment):
    # Base case: if all variables are assigned, return
    if len(assignment) == len(variables):
        return assignment

    # Select an unassigned variable
    var = [v for v in variables if v not in assignment][0]
    for value in domains[var]:
        if is_valid(assignment, var, value):
            assignment[var] = value
            result = backtracking(assignment)
            if result:
                return result
            assignment.pop(var)  # Backtrack

    return None

solution = backtracking({})
```

variables = ['A', 'B', 'C', 'D', 'E', 'F', 'G']

domains = {'A': ['Red', 'Green', 'Blue'], 'B': ['Red', 'Green', 'Blue'], ... }

assignment = {}

var = A
value = 'Red'



17

Backtracking: Trace

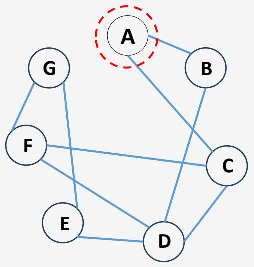
```
def backtracking(assignment):
    # Base case: if all variables are assigned, return
    if len(assignment) == len(variables):
        return assignment

    # Select an unassigned variable
    var = [v for v in variables if v not in assignment][0]
    for value in domains[var]:
        if is_valid(assignment, var, value): ✓
            assignment[var] = value
            result = backtracking(assignment)
            if result:
                return result
            assignment.pop(var)  # Backtrack

    return None

solution = backtracking({})
```

variables = ['A', 'B', 'C', 'D', 'E', 'F', 'G']

domains = {'A': ['Red', 'Green', 'Blue'], 'B': ['Red', 'Green', 'Blue'], ... }

assignment = {}

var = A
value = 'Red'



18

Backtracking: Trace

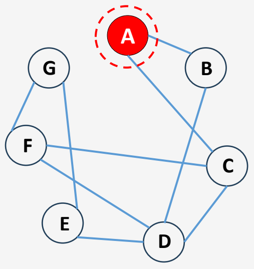
```
def backtracking(assignment):
    # Base case: if all variables are assigned, return
    if len(assignment) == len(variables):
        return assignment

    # Select an unassigned variable
    var = [v for v in variables if v not in assignment][0]
    for value in domains[var]:
        if is_valid(assignment, var, value):
            assignment[var] = value
            result = backtracking(assignment)
            if result:
                return result
            assignment.pop(var)  # Backtrack

    return None

solution = backtracking({})
```

variables = ['A', 'B', 'C', 'D', 'E', 'F', 'G']

domains = {'A': ['Red', 'Green', 'Blue'], 'B': ['Red', 'Green', 'Blue'], ... }

assignment = {'A': 'Red'}

var = A
value = 'Red'



CENG-3511 Artificial Intelligence

19

Backtracking: Trace

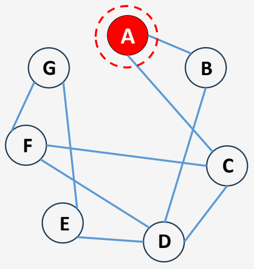
```
def backtracking(assignment):
    # Base case: if all variables are assigned, return
    if len(assignment) == len(variables):
        return assignment

    # Select an unassigned variable
    var = [v for v in variables if v not in assignment][0]
    for value in domains[var]:
        if is_valid(assignment, var, value):
            assignment[var] = value
            result = backtracking(assignment)
            if result:
                return result
            assignment.pop(var)  # Backtrack

    return None

solution = backtracking({})
```

variables = ['A', 'B', 'C', 'D', 'E', 'F', 'G']

domains = {'A': ['Red', 'Green', 'Blue'], 'B': ['Red', 'Green', 'Blue'], ... }

assignment = {'A': 'Red'}

var = A
value = 'Red'



CENG-3511 Artificial Intelligence

20

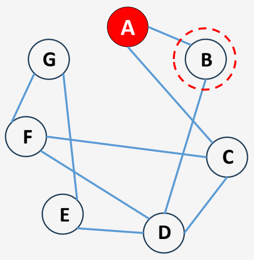
Backtracking: Trace

```python
def backtracking(assignment):
    # Base case: if all variables are assigned, return
    if len(assignment) == len(variables):
        return assignment

    # Select an unassigned variable
    var = [v for v in variables if v not in assignment][0]
    for value in domains[var]:
        if is_valid(assignment, var, value):
            assignment[var] = value
            result = backtracking(assignment)
            if result:
                return result
            assignment.pop(var)  # Backtrack

    return None

solution = backtracking({})
```

variables = ['A', 'B', 'C', 'D', 'E', 'F', 'G']
domains = {'A': ['Red', 'Green', 'Blue'], 'B': ['Red', 'Green', 'Blue'], … }

assignment = {'A': 'Red'}

var = B
value = 'Red'



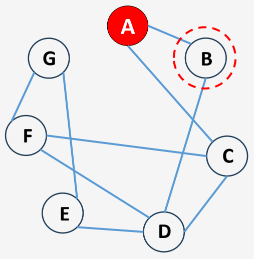
Backtracking: Trace

```python
def backtracking(assignment):
    # Base case: if all variables are assigned, return
    if len(assignment) == len(variables):
        return assignment

    # Select an unassigned variable
    var = [v for v in variables if v not in assignment][0]
    for value in domains[var]:
        if is_valid(assignment, var, value):
            assignment[var] = value
            result = backtracking(assignment)
            if result:
                return result
            assignment.pop(var)  # Backtrack

    return None

solution = backtracking({})
```

variables = ['A', 'B', 'C', 'D', 'E', 'F', 'G']
domains = {'A': ['Red', 'Green', 'Blue'], 'B': ['Red', 'Green', 'Blue'], … }

assignment = {'A': 'Red'}

var = B
value = 'Red'



Backtracking: Trace

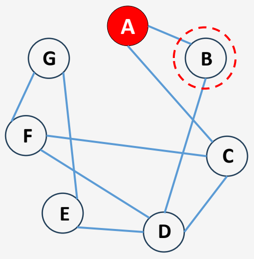
```
def backtracking(assignment):
    # Base case: if all variables are assigned, return
    if len(assignment) == len(variables):
        return assignment

    # Select an unassigned variable
    var = [v for v in variables if v not in assignment][0]
    for value in domains[var]:
        if is_valid(assignment, var, value): ❌
            assignment[var] = value
            result = backtracking(assignment)
            if result:
                return result
            assignment.pop(var)  # Backtrack

    return None

solution = backtracking({})
```

variables = ['A', 'B', 'C', 'D', 'E', 'F', 'G']

domains = {'A': ['Red', 'Green', 'Blue'], 'B': ['Red', 'Green', 'Blue'], ... }

assignment = {'A': 'Red'}

var = B
value = 'Red'



Backtracking: Trace

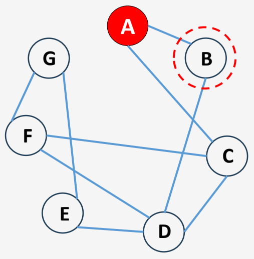
```
def backtracking(assignment):
    # Base case: if all variables are assigned, return
    if len(assignment) == len(variables):
        return assignment

    # Select an unassigned variable
    var = [v for v in variables if v not in assignment][0]
    for value in domains[var]:
        if is_valid(assignment, var, value):
            assignment[var] = value
            result = backtracking(assignment)
            if result:
                return result
            assignment.pop(var)  # Backtrack

    return None

solution = backtracking({})
```

variables = ['A', 'B', 'C', 'D', 'E', 'F', 'G']

domains = {'A': ['Red', 'Green', 'Blue'], 'B': ['Red', 'Green', 'Blue'], ... }

assignment = {'A': 'Red'}

var = B
value = 'Green'

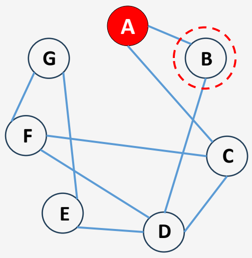


Backtracking: Trace

```
def backtracking(assignment):
    # Base case: if all variables are assigned, return
    if len(assignment) == len(variables):
        return assignment

    # Select an unassigned variable
    var = [v for v in variables if v not in assignment][0]
    for value in domains[var]:
        if is_valid(assignment, var, value): ✓
            assignment[var] = value
            result = backtracking(assignment)
            if result:
                return result
            assignment.pop(var)  # Backtrack

    return None

solution = backtracking({})
```

variables = ['A', 'B', 'C', 'D', 'E', 'F', 'G']
domains = {'A': ['Red', 'Green', 'Blue'], 'B': ['Red', 'Green', 'Blue'], … }
assignment = {'A': 'Red'}

var = B
value = 'Green'



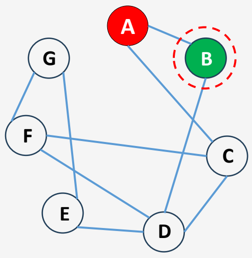
25

Backtracking: Trace

```
def backtracking(assignment):
    # Base case: if all variables are assigned, return
    if len(assignment) == len(variables):
        return assignment

    # Select an unassigned variable
    var = [v for v in variables if v not in assignment][0]
    for value in domains[var]:
        if is_valid(assignment, var, value):
            assignment[var] = value
            result = backtracking(assignment)
            if result:
                return result
            assignment.pop(var)  # Backtrack

    return None

solution = backtracking({})
```

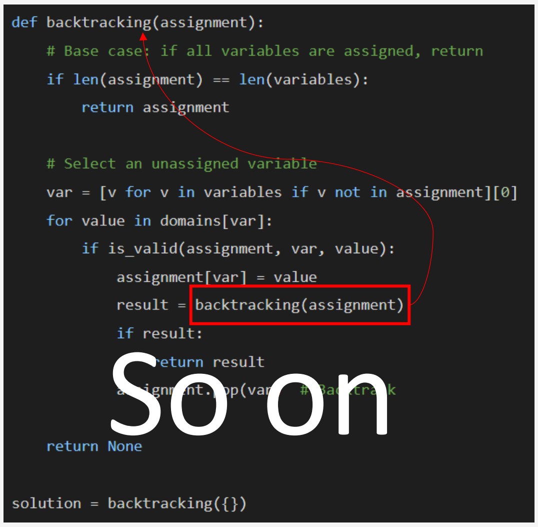
variables = ['A', 'B', 'C', 'D', 'E', 'F', 'G']
domains = {'A': ['Red', 'Green', 'Blue'], 'B': ['Red', 'Green', 'Blue'], … }
assignment = {'A': 'Red', 'B': 'Green'}

var = B
value = 'Green'



26

Backtracking: Trace



```
def backtracking(assignment):
    # Base case: if all variables are assigned, return
    if len(assignment) == len(variables):
        return assignment

    # Select an unassigned variable
    var = [v for v in variables if v not in assignment][0]
    for value in domains[var]:
        if is_valid(assignment, var, value):
            assignment[var] = value
            result = backtracking(assignment)
            if result:
                return result
            assignment.pop(var)  # Backtrack
    return None

solution = backtracking({})
```

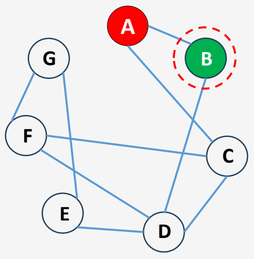
So on

variables = ['A', 'B', 'C', 'D', 'E', 'F', 'G']

domains = {'A': ['Red', 'Green', 'Blue'], 'B': ['Red', 'Green', 'Blue'], ... }

assignment = {'A': 'Red', 'B': 'Green'}

var = B
value = 'Green'



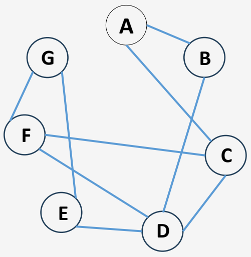
27

Search Tree for Map Coloring Problem & Backtracking

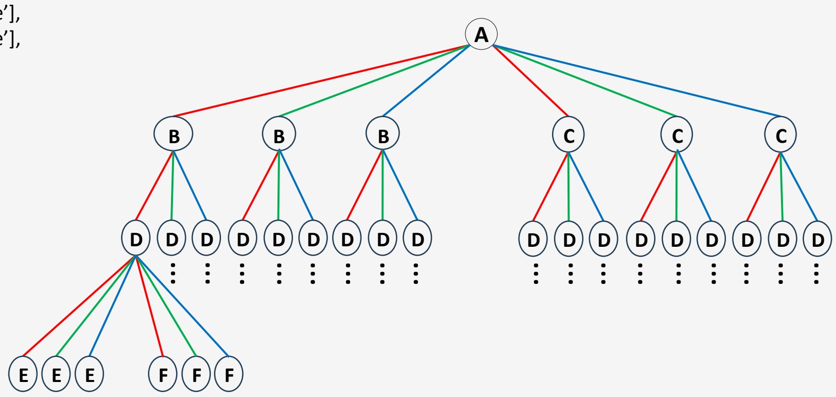
variables = ['A', 'B', 'C', 'D', 'E', 'F', 'G']

domains = {'A': ['Red', 'Green', 'Blue'],
'B': ['Red', 'Green', 'Blue'],
... }

Graph Representation
(Model of the Problem)



Search Tree/Space

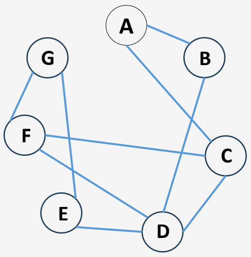


28

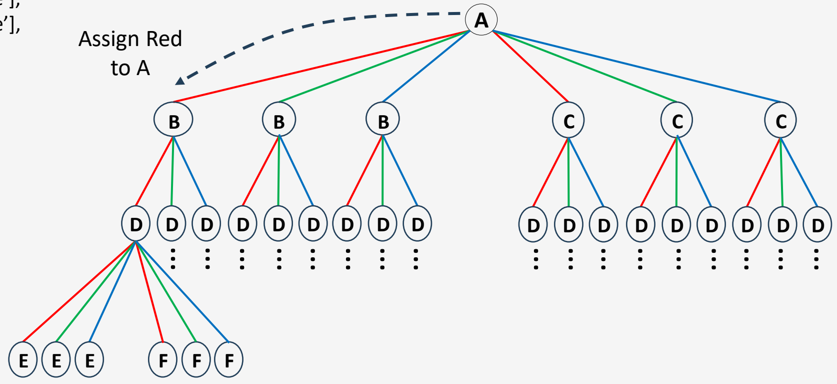
Search Tree for Map Coloring Problem & Backtracking

variables = ['A', 'B', 'C', 'D', 'E', 'F', 'G']
domains = {'A': ['Red', 'Green', 'Blue'],
'B': ['Red', 'Green', 'Blue'],
… }

Graph Representation
(Model of the Problem)



State: A is selected, 'Red' is being checked

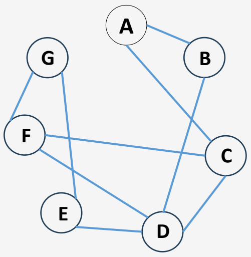


29

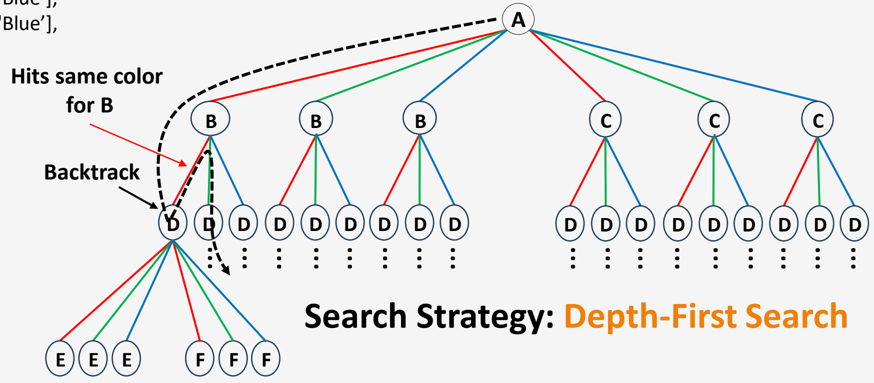
Search Tree for Map Coloring Problem & Backtracking

variables = ['A', 'B', 'C', 'D', 'E', 'F', 'G']
domains = {'A': ['Red', 'Green', 'Blue'],
'B': ['Red', 'Green', 'Blue'],
… }

Graph Representation
(Model of the Problem)



State: B is selected, 'Red' is being checked



Search Strategy: Depth-First Search

30

Backtracking: Observation

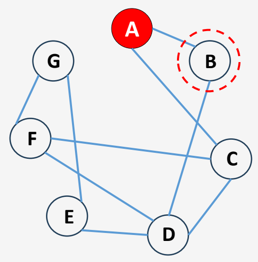
State: B is selected, ‘Red’ is being checked

```
def backtracking(assignment):
    # Base case: if all variables are assigned, return
    if len(assignment) == len(variables):
        return assignment

    # Select an unassigned variable
    var = [v for v in variables if v not in assignment][0]
    for value in domains[var]:
        if is_valid(assignment, var, value): ❌
            assignment[var] = value
            result = backtracking(assignment)
            if result:
                return result
            assignment.pop(var)  # Backtrack

    return None

solution = backtracking({})
```

Can we do better?

variables = ['A', 'B', 'C', 'D', 'E', 'F', 'G']
domains = {'A': ['Red', 'Green', 'Blue'], 'B': ['Red', 'Green', 'Blue'], … }
assignment = {'A': 'Red'}

var = B
value = 'Red'



31

Backtracking: Observation

State: B is selected, ‘Red’ is being checked

```
def backtracking(assignment):
    # Base case: if all variables are as
    if len(assignment) == len(variables):
        return assignment

    # Select an unassigned variable
    var = [v for v in variables if v not in assignment][0
    for value in domains[var]:
        if is_valid(assignment, var, value): ❌
            assignment[var] = value
            result = backtracking(assignment)
            if result:
                return result
            assignment.pop(var)  # Backtrack

    return None

solution = backtracking({})
```

Can we do better?

variables = ['A', 'B', 'C', 'D', 'E', 'F', 'G']
domains = {'A': ['Red', 'Green', 'Blue'], 'B': ['Red', 'Green', 'Blue'], … }
assignment = {'A': 'Red'}

var = B
value = 'Red'

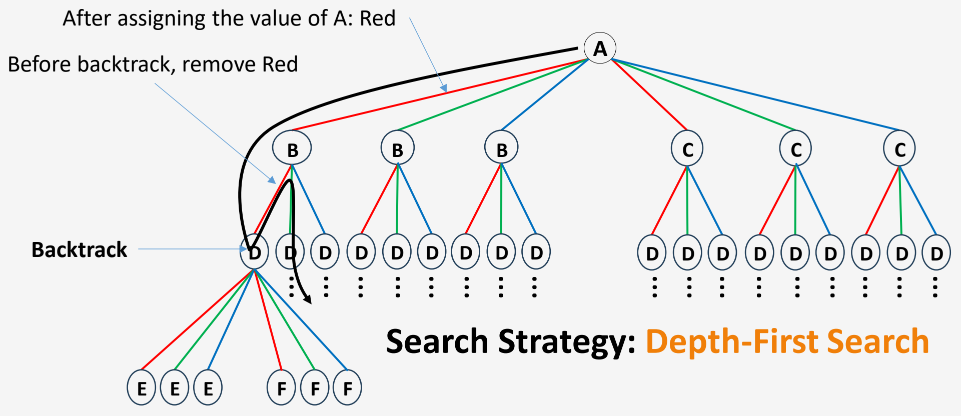
Yes
We can adjust the domains of neighboring nodes

After assignment, we can “forward check” the domains of the neighboring variables (nodes) and remove the value (color) of the currently assigned variable.

That is, when A is assigned to Red, we can “forward check” the domains of the neighboring variables (B) and remove the value (Red) from the domain of neighboring variables (B).

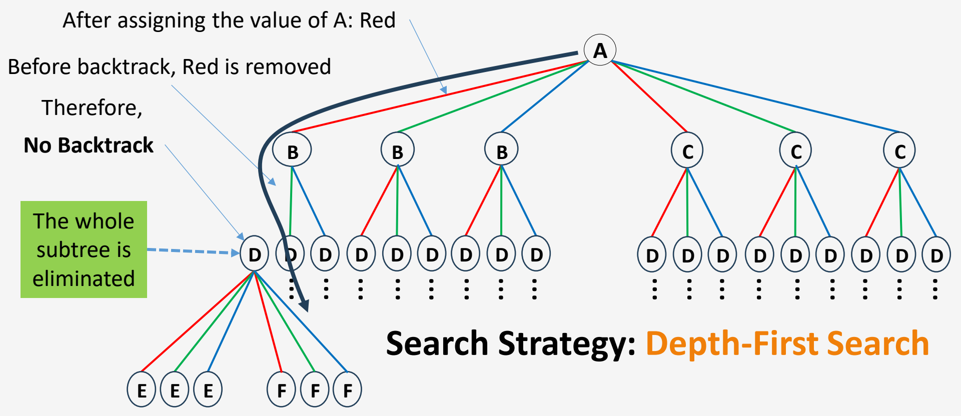
32

Backtracking: Observation


After assigning the value of A: Red
Before backtrack, remove Red
Backtrack
A
B
B
B
C
C
C
D
E
F

Search Strategy: Depth-First Search

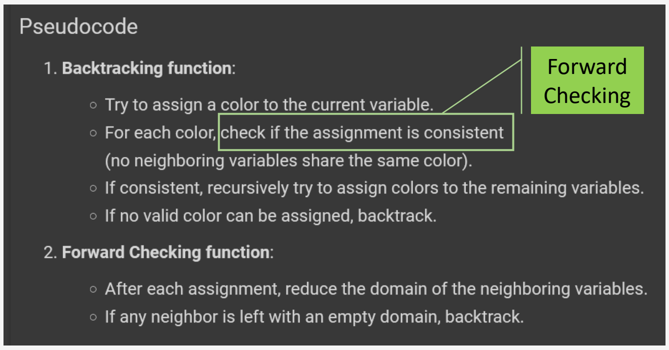
Backtracking: Observation


After assigning the value of A: Red
Before backtrack, Red is removed
Therefore,
No Backtrack
The whole subtree is eliminated
A
B
B
B
C
C
C
D
E
F

Search Strategy: Depth-First Search

Forward Checking

CENG3511-AI-Lab3-ContrainSatisfactionProblems.ipynb



Pseudocode

```
1. Backtracking function:
    ◦ Try to assign a color to the current variable.
    ◦ For each color, check if the assignment is consistent
      (no neighboring variables share the same color).
    ◦ If consistent, recursively try to assign colors to the remaining variables.
    ◦ If no valid color can be assigned, backtrack.
2. Forward Checking function:
    ◦ After each assignment, reduce the domain of the neighboring variables.
    ◦ If any neighbor is left with an empty domain, backtrack.
```

Forward Checking

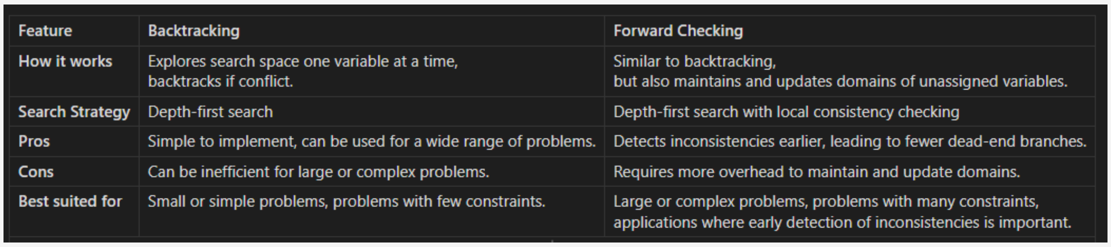
Idea: As a variable (city) is assigned a value (color), update the domains (color list) of neighboring variables (cities).

Rationale: This allows for early detection of inconsistencies and pruning of branches.

Example: If a region (variable) is assigned a color (value) that conflicts with a neighbor's domain (color list), the branch can be immediately pruned.

Exercise (colab notebook)
Compare the performance (time and nodes explored) of Backtracking with and without Forward Checking.

Backtracking vs Forward Checking



Feature	Backtracking	Forward Checking
How it works	Explores search space one variable at a time, backtracks if conflict.	Similar to backtracking, but also maintains and updates domains of unassigned variables.
Search Strategy	Depth-first search	Depth-first search with local consistency checking
Pros	Simple to implement, can be used for a wide range of problems.	Detects inconsistencies earlier, leading to fewer dead-end branches.
Cons	Can be inefficient for large or complex problems.	Requires more overhead to maintain and update domains.
Best suited for	Small or simple problems, problems with few constraints.	Large or complex problems, problems with many constraints, applications where early detection of inconsistencies is important.

Ordering Strategies in Backtracking

- **Backtracking** is a general algorithm for finding solutions to combinatorial problems.
  - It involves exploring a search tree, where each node represents a partial solution.
- **To improve the efficiency of backtracking**, various ordering strategies can be employed, such that
  - Domain Filtering (e.g., **Forward Checking**)
  - Minimum Remaining Values (MRV)
  - Degree Heuristic
  - Least Constraining Value (LCV)
  - Constraint Propagation

Ordering Strategies in Backtracking

Domain Filtering (Forward Checking)

- **Idea**: As a variable is assigned a value, update the domains of neighboring variables.
- **Rationale**: This allows for early detection of inconsistencies and pruning of branches.
- **Example**: If a variable is assigned a value that conflicts with a neighbor's domain, the branch can be immediately pruned.

Ordering Strategies in Backtracking

Minimum Remaining Values (MRV)

- **Idea**: Choose the node with the fewest remaining possible values..
- **Rationale**: Nodes with fewer options are more likely to lead to dead ends.
- **Example**: If one variable has only two possible values, it's more efficient to assign it first to see if it leads to a solution.

Ordering Strategies in Backtracking

Degree Heuristic

- **Idea**: Choose the node with the most constraints on other variables.
- **Rationale**: By focusing on variables with many constraints, we can quickly identify conflicts and prune branches.
- **Example**: A variable that is involved in many constraints with other variables is more likely to affect the overall solution.

Ordering Strategies in Backtracking

Least Constraining Value (LCV)

- **Idea**: Choose the value that rules out the fewest values for neighboring variables.
- **Rationale**: This strategy helps to minimize the number of dead ends encountered later in the search.
- **Example**: If one value for a variable eliminates many possibilities for other variables, it's better to try a value that is less restrictive.

Ordering Strategies in Backtracking

Constraint Propagation

- **Idea**: Use constraint propagation techniques (e.g., **arc consistency**) to reduce the domains of variables.
- **Rationale**: This can significantly reduce the search space and improve efficiency.
- **Example**: Arc consistency ensures that for every pair of variables connected by a constraint, the domain of one variable contains a value that is consistent with every value in the domain of the other variable.
- **Common Techniques**
  - **Arc consistency**: Ensures that for every pair of variables connected by a constraint, the domain of one variable contains a value that is consistent with every value in the domain of the other variable.
  - **Path consistency**: Ensures that for every pair of variables connected by a path of constraints, there exists a consistent assignment for the variables along that path.
  - **k-consistency**: A generalization of arc and path consistency, ensuring that for every subset of k variables, there exists a consistent assignment for those variables.

Advance Techniques

Arc Consistency and Min-Conflicts

CENG-3511 Artificial Intelligence

44

Arc Consistency (AC-3)

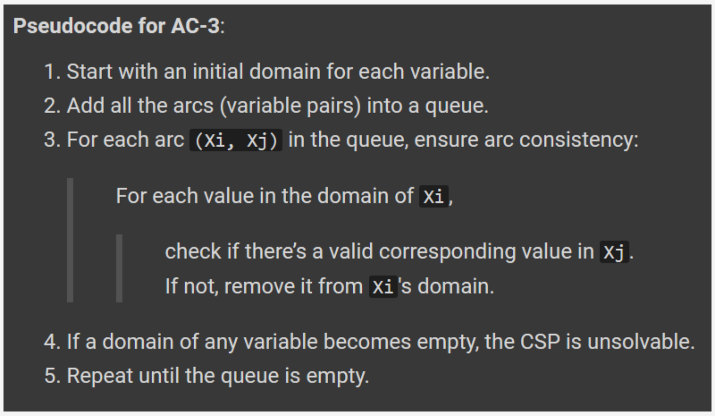
CENG3511-AI-Lab3-ContrainSatisfactionProblems.ipynb

Definition

A variable is arc-consistent, if every value in its domain satisfies the constraints with its neighbors.

AC-3 Algorithm:

Removes values from variable domains if they do not satisfy the binary constraints with neighboring variables.



Pseudocode for AC-3:

```
1. Start with an initial domain for each variable.
2. Add all the arcs (variable pairs) into a queue.
3. For each arc (Xi, Xj) in the queue, ensure arc consistency:
       For each value in the domain of Xi,
           check if there's a valid corresponding value in Xj.
           If not, remove it from Xi's domain.
4. If a domain of any variable becomes empty, the CSP is unsolvable.
5. Repeat until the queue is empty.
```

CENG-3511 Artificial Intelligence

45

Min-Conflict

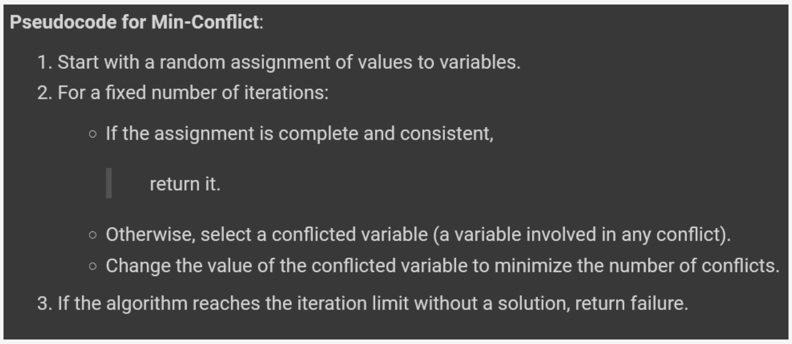
CENG3511-AI-Lab3-ContrainSatisfactionProblems.ipynb

Definition

A heuristic algorithm that starts with an initial (random) solution and iteratively fixes conflicts by selecting the value with the fewest conflicts.

Useful:

when quick, approximate solutions are needed, especially for large CSPs, though it doesn't guarantee **completeness**



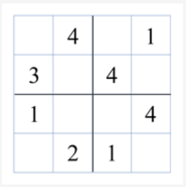
Pseudocode for Min-Conflict:

```
1. Start with a random assignment of values to variables.
2. For a fixed number of iterations:
     ○ If the assignment is complete and consistent,
         return it.
     ○ Otherwise, select a conflicted variable (a variable involved in any conflict).
     ○ Change the value of the conflicted variable to minimize the number of conflicts.
3. If the algorithm reaches the iteration limit without a solution, return failure.
```

Example: Sudoku

CENG3511-AI-Lab3-Sudoku.ipynb

A 2x2 sub-grid Sudoku (or a 4x4 Sudoku)



	4		1
3		4	
1			4
	2	1	

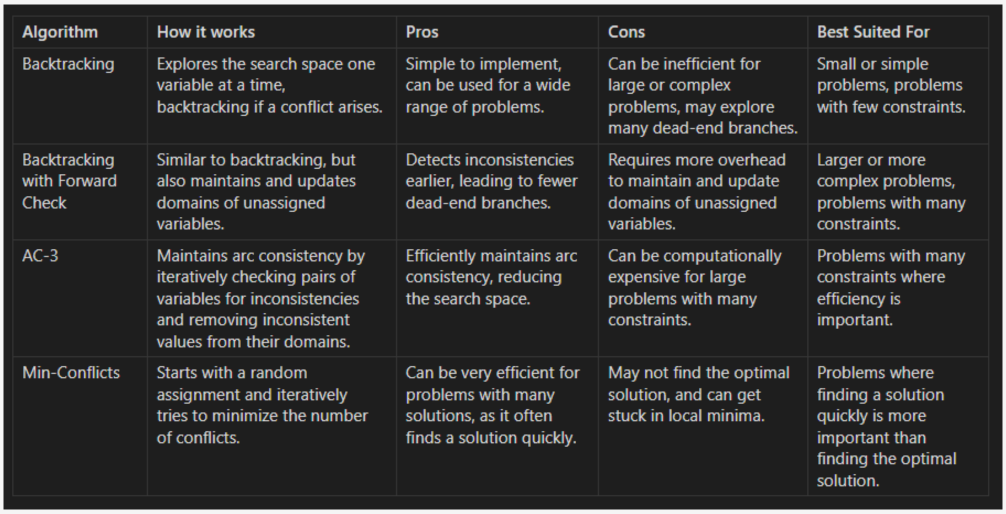
Constraints:
Each row , **Each column**, and **Each 2x2 sub-grid** must contain the numbers 1 to 4 exactly once.

A Possible Solution Strategy

1. **Backtracking Search**:
- Assign numbers to the empty cells, one by one, in a depth-first manner.
- After each assignment, ensure that the assignment doesn't violate Sudoku rules (i.e., uniqueness in the row, column, and 2x2 sub-grid).
- If the assignment leads to a dead-end (i.e., no valid assignments left), backtrack.

2. **Forward Checking**:
- After assigning a number to a cell, update the domains (valid possible numbers) for the unassigned cells in the same row, column, and sub-grid.
- If any cell is left with an empty domain after an assignment, backtrack immediately.

Summary: Comparison of Backtracking Algorithms



Algorithm	How it works	Pros	Cons	Best Suited For
Backtracking	Explores the search space one variable at a time, backtracking if a conflict arises.	Simple to implement, can be used for a wide range of problems.	Can be inefficient for large or complex problems, may explore many dead-end branches.	Small or simple problems, problems with few constraints.
Backtracking with Forward Check	Similar to backtracking, but also maintains and updates domains of unassigned variables.	Detects inconsistencies earlier, leading to fewer dead-end branches.	Requires more overhead to maintain and update domains of unassigned variables.	Larger or more complex problems, problems with many constraints.
AC-3	Maintains arc consistency by iteratively checking pairs of variables for inconsistencies and removing inconsistent values from their domains.	Efficiently maintains arc consistency, reducing the search space.	Can be computationally expensive for large problems with many constraints.	Problems with many constraints where efficiency is important.
Min-Conflicts	Starts with a random assignment and iteratively tries to minimize the number of conflicts.	Can be very efficient for problems with many solutions, as it often finds a solution quickly.	May not find the optimal solution, and can get stuck in local minima.	Problems where finding a solution quickly is more important than finding the optimal solution.

Adversarial Search

Minimax, Apha-Beta Pruning and Heuristic Evaluation

Adversarial Search and Game Theory

Definition:

Adversarial search is used in games or situations where multiple agents (players) compete directly, and their interests are opposed.

Goal:

One agent's gain is another agent's loss (zero-sum)

Tic-Tac-Toe

Connect-4

Chess

Finance

Two-Player Zero-Sum Games

Definition

A type of game where one player's loss is exactly the other's gain.

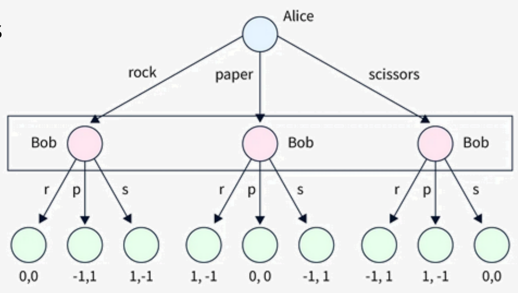
Formal Representation

- Two players:
  - **MAX** (tries to maximize score) and
  - **MIN** (tries to minimize score).
- Utility values:
  - Represent the final outcome (e.g., **win**, **lose**, or **draw**).
- Assumptions
  - **Both players play optimally**.
  - Each move brings the game closer to a terminal state (win/lose/draw).

Game Representation: Search Trees

Games are modeled as search trees:

- Nodes represent game states.
- Edges represent possible moves.
- Leaf nodes represent terminal states (win/lose/draw).



Minimax Algorithm: Concept

Minimax is a recursive algorithm (like Backtracking) for choosing the best move in two-player zero-sum games.

Key Idea:

- **MAX** tries to maximize the value of the game state.
- **MIN** tries to minimize it.
- **Both players assume** that the opponent will also play optimally.

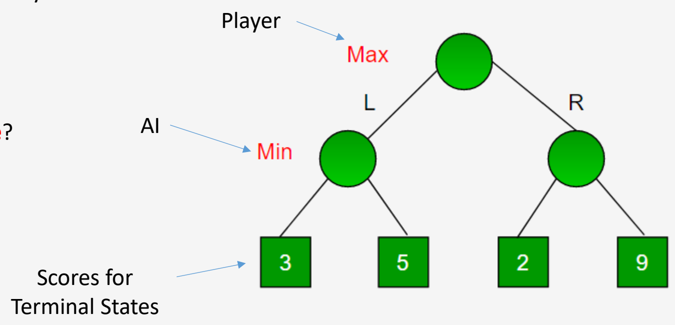
Minimax Decision Rule:

"Choose the move that minimizes the opponent's maximum payoff."

Example

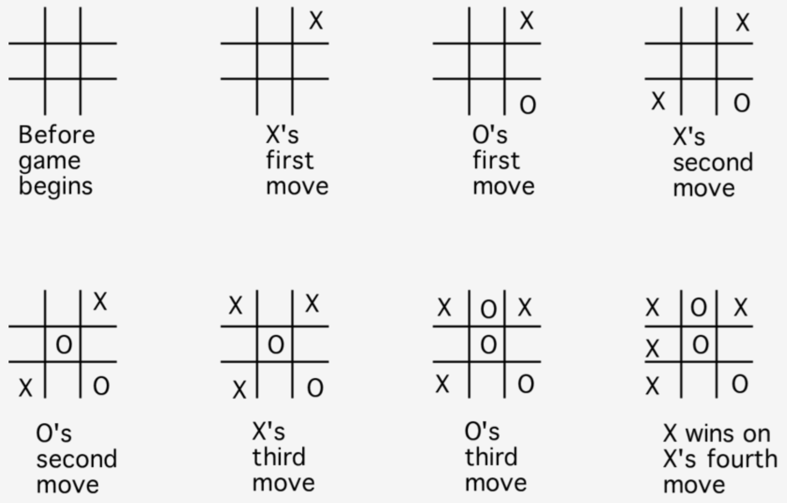
- **Player** will play to get the **max** score
- **AI** will play to get the **min** score to Player

It is the Player's turn.
What would be the **best move**?



54

Example: Tic-Tac-Toe



55

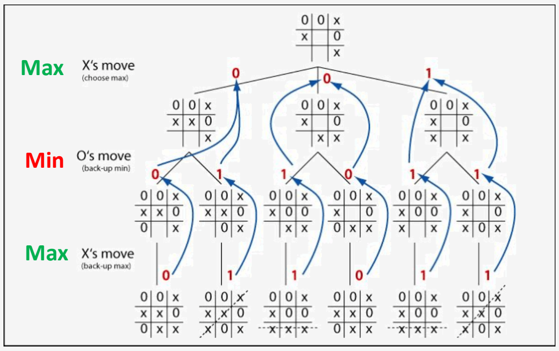
Example: Tic-Tac-Toe using Minimax

Step 1: Build the game tree with all possible future moves.

Step 2: Evaluate the leaf nodes (terminal states: win, lose, draw).

Step 3: Use the Minimax rule to backpropagate the utility values.

Step 4: MAX chooses the move that maximizes the minimum gain (assuming MIN will play optimally).

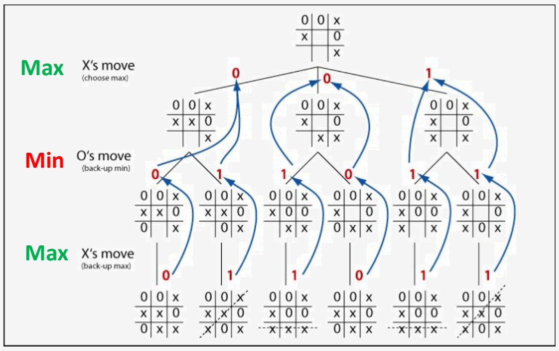


Example: Tic-Tac-Toe using Minimax

Pseudo Code for MiniMax

```
def minimax(state, depth, maximizingPlayer):
    if terminal(state) or depth == 0:
        return evaluate(state)

    if maximizingPlayer:
        maxEval = -∞
        for each child in state:
            eval = minimax(child, depth - 1, False)
            maxEval = max(maxEval, eval)
        return maxEval
    else:
        minEval = ∞
        for each child in state:
            eval = minimax(child, depth - 1, True)
            minEval = min(minEval, eval)
        return minEval
```

CENG3511-AI-Lab3-Advisarial Search and Games.ipynb



Assume optimal playing for both opponent

Purpose of the evaluate(state) Function

- It assigns a numerical value to a game state.
- Determines how favorable or unfavorable a position is for a given player.

Why is it important?

- In non-terminal states (not win/loss/draw), it estimates the potential outcome.
- Helps guide the Minimax decision-making process when the game is still ongoing.

When is evaluate(state) Called?

- **Terminal State**: If the game has reached a terminal state (win/loss/draw), a final utility score is returned (e.g., +1 for a win, -1 for a loss, 0 for a draw).

- **Non-Terminal State**: When the game hasn't reached a final state, evaluate(state) estimates the value of the current state based on board conditions.

Key Components of the evaluate(state) Function

Game-Specific: The evaluation function varies based on the game (e.g., Chess, Tic-Tac-Toe).

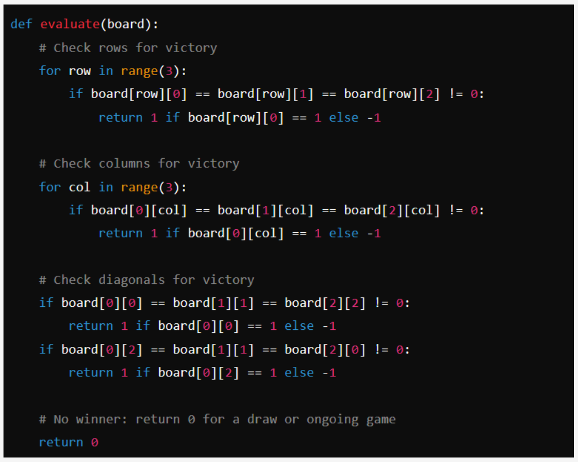
- **Material Evaluation**: Counts the number of pieces for each player (Chess).
- **Winning Potential**: Looks at potential win/loss scenarios (Tic-Tac-Toe).

Numeric Output:

- **Positive score** (e.g., +1): Advantageous for the MAX player.
- Negative score (e.g., -1): Advantageous for the MIN player.
- **Zero score** : Neutral or equal advantage for both players.

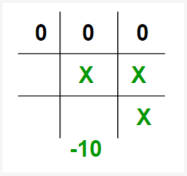
60

Example: Tic-Tac-Toe

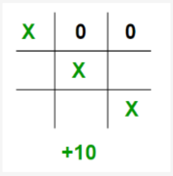


```
def evaluate(board):
    # Check rows for victory
    for row in range(3):
        if board[row][0] == board[row][1] == board[row][2] != 0:
            return 1 if board[row][0] == 1 else -1

    # Check columns for victory
    for col in range(3):
        if board[0][col] == board[1][col] == board[2][col] != 0:
            return 1 if board[0][col] == 1 else -1

    # Check diagonals for victory
    if board[0][0] == board[1][1] == board[2][2] != 0:
        return 1 if board[0][0] == 1 else -1
    if board[0][2] == board[1][1] == board[2][0] != 0:
        return 1 if board[0][2] == 1 else -1

    # No winner: return 0 for a draw or ongoing game
    return 0
```

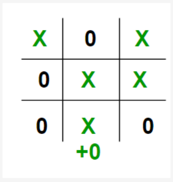
Row: O Wins



Diagonal: X Wins



Draw/Tie



61

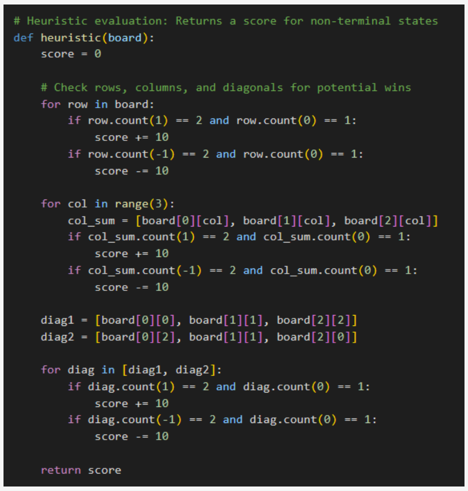
Heuristic Evaluation Functions

In non-terminal states, apply **heuristic evaluation** to estimate how favorable a state is for MAX or MIN.

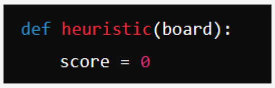
Potential Heuristics:

- Number of winning lines still open for each player.
- Number of 2-in-a-row configurations with an open third spot.
- Prioritize corners or center control in Tic-Tac-Toe.

Heuristic Evaluation Functions: Tic-Tac-Toe



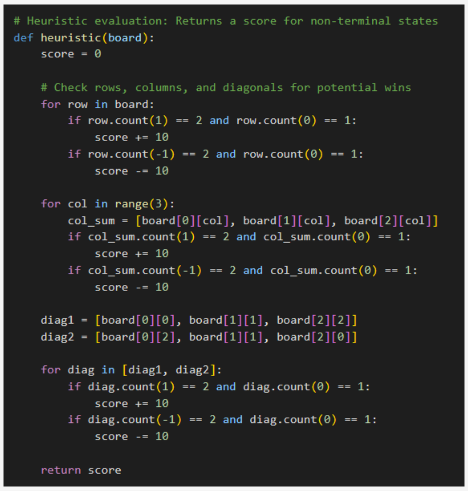
```python
# Heuristic evaluation: Returns a score for non-terminal states
def heuristic(board):
    score = 0

    # Check rows, columns, and diagonals for potential wins
    for row in board:
        if row.count(1) == 2 and row.count(0) == 1:
            score += 10
        if row.count(-1) == 2 and row.count(0) == 1:
            score -= 10

    for col in range(3):
        col_sum = [board[0][col], board[1][col], board[2][col]]
        if col_sum.count(1) == 2 and col_sum.count(0) == 1:
            score += 10
        if col_sum.count(-1) == 2 and col_sum.count(0) == 1:
            score -= 10

    diag1 = [board[0][0], board[1][1], board[2][2]]
    diag2 = [board[0][2], board[1][1], board[2][0]]

    for diag in [diag1, diag2]:
        if diag.count(1) == 2 and diag.count(0) == 1:
            score += 10
        if diag.count(-1) == 2 and diag.count(0) == 1:
            score -= 10

    return score
```



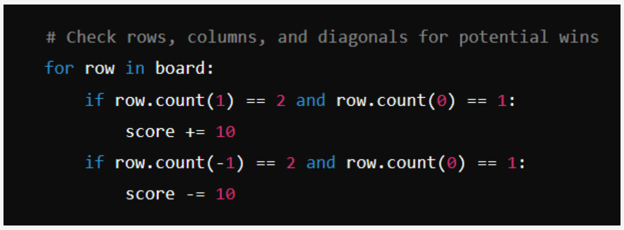
```python
def heuristic(board):
    score = 0
```

The score variable keeps track of how advantageous the board is for the current player. A higher score is favorable for player 1 (denoted by 1), while a lower score is favorable for player 2 (denoted by -1).

Heuristic Evaluation Functions: Tic-Tac-Toe



```python
# Heuristic evaluation: Returns a score for non-terminal states
def heuristic(board):
    score = 0

    # Check rows, columns, and diagonals for potential wins
    for row in board:
        if row.count(1) == 2 and row.count(0) == 1:
            score += 10
        if row.count(-1) == 2 and row.count(0) == 1:
            score -= 10

    for col in range(3):
        col_sum = [board[0][col], board[1][col], board[2][col]]
        if col_sum.count(1) == 2 and col_sum.count(0) == 1:
            score += 10
        if col_sum.count(-1) == 2 and col_sum.count(0) == 1:
            score -= 10

    diag1 = [board[0][0], board[1][1], board[2][2]]
    diag2 = [board[0][2], board[1][1], board[2][0]]

    for diag in [diag1, diag2]:
        if diag.count(1) == 2 and diag.count(0) == 1:
            score += 10
        if diag.count(-1) == 2 and diag.count(0) == 1:
            score -= 10

    return score
```



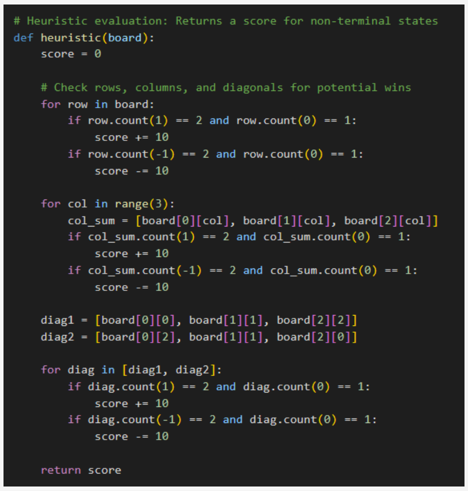
```python
# Check rows, columns, and diagonals for potential wins
for row in board:
    if row.count(1) == 2 and row.count(0) == 1:
        score += 10
    if row.count(-1) == 2 and row.count(0) == 1:
        score -= 10
```

For each row in the board:

- The function checks if player 1 has two markers (1) and there is one empty space (0).
- If this condition is met, the function adds 10 to the score, as this represents a strong potential for player 1 to win in the next move.
- Similarly, it checks if player 2 has two markers (-1) and one empty space. If true, 10 points are subtracted from the score, since this is a strong opportunity for player 2 to win in the next move.

64

Heuristic Evaluation Functions: Tic-Tac-Toe



```python
# Heuristic evaluation: Returns a score for non-terminal states
def heuristic(board):
    score = 0

    # Check rows, columns, and diagonals for potential wins
    for row in board:
        if row.count(1) == 2 and row.count(0) == 1:
            score += 10
        if row.count(-1) == 2 and row.count(0) == 1:
            score -= 10

    for col in range(3):
        col_sum = [board[0][col], board[1][col], board[2][col]]
        if col_sum.count(1) == 2 and col_sum.count(0) == 1:
            score += 10
        if col_sum.count(-1) == 2 and col_sum.count(0) == 1:
            score -= 10

    diag1 = [board[0][0], board[1][1], board[2][2]]
    diag2 = [board[0][2], board[1][1], board[2][0]]

    for diag in [diag1, diag2]:
        if diag.count(1) == 2 and diag.count(0) == 1:
            score += 10
        if diag.count(-1) == 2 and diag.count(0) == 1:
            score -= 10

    return score
```

Heuristic Logic:

The function is evaluating positions based on immediate winning opportunities (two markers and one empty space). The higher the score, the closer player 1 is to winning. If player 2 is closer to winning, the score is driven negative.

Example Scenarios:

- A row like [1, 1, 0] (two X's and an empty space) would increase the score by 10 because player 1 has a potential to win in the next move.
- A row like [-1, -1, 0] (two O's and an empty space) would decrease the score by 10, signaling that player 2 is close to winning.

65

Limitations of Minimax

- **Computational Complexity**: Searching all possible moves grows exponentially with depth, i.e., $O(b^d)$.
- **Game Tree Size**: For large games like Chess, the tree is enormous.
- **Optimal Play Assumption**: Assumes the opponent plays optimally at all times.

66

Improvement: Alpha-Beta Pruning

Alpha-Beta Pruning:

An optimization to the Minimax algorithm that reduces the number of nodes evaluated.

- Prune (ignore) branches that don't affect the final decision.

Key Idea:

- Prune branches where a move is already proven worse than a previously explored option.

67

Concept of Alpha-Beta Pruning

- **Alpha**: The best value that the MAX player can guarantee so far (typically -∞).
- **Beta**: The best value that the MIN player can guarantee so far (typically +∞).
- During the traversal of the game tree:
  - if a node's value (or potential value) makes it impossible to improve on an already found value for either player, we prune that branch, avoiding unnecessary calculations.
  - That is, **if beta <= alpha**

Alpha-Beta Pruning Process

The Minimax algorithm is still used to compute the best possible move, but two variables—alpha and beta—are introduced:

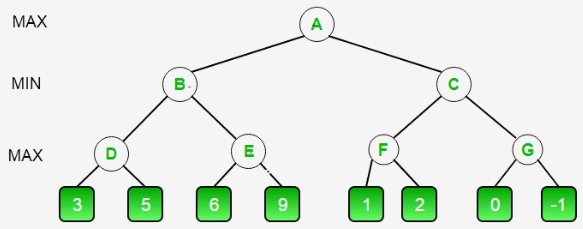
- Alpha is updated during the MAX player's turn to track the highest score found so far.
- Beta is updated during the MIN player's turn to track the lowest score found so far.

As the tree is traversed,
if the current node cannot improve
on alpha (for the MAX player) or beta (for the MIN player),
that branch is cut off (pruned).
Beta <= Alpha

Example

```
function minimax(node, depth, isMaximizingPlayer, alpha, beta):

    if node is a leaf node :
        return value of the node

    if isMaximizingPlayer :
        bestVal = -INFINITY
        for each child node :
            value = minimax(node, depth+1, false, alpha, beta)
            bestVal = max( bestVal, value)
            alpha = max( alpha, bestVal)
            if beta <= alpha:
                break
        return bestVal

    else :
        bestVal = +INFINITY
        for each child node :
            value = minimax(node, depth+1, true, alpha, beta)
            bestVal = min( bestVal, value)
            beta = min( beta, bestVal)
            if beta <= alpha:
                break
        return bestVal
```

```
// Calling the function for the first time.
minimax(0, 0, true, -INFINITY, +INFINITY)
```

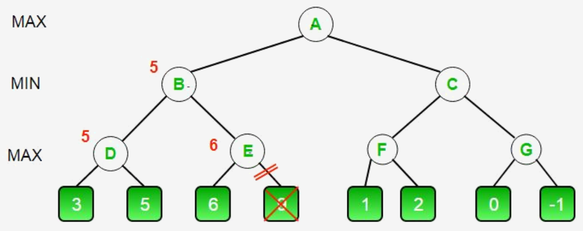


70

Example

```
function minimax(node, depth, isMaximizingPlayer, alpha, beta):

    if node is a leaf node :
        return value of the node

    if isMaximizingPlayer :
        bestVal = -INFINITY
        for each child node :
            value = minimax(node, depth+1, false, alpha, beta)
            bestVal = max( bestVal, value)
            alpha = max( alpha, bestVal)
            if beta <= alpha:
                break
        return bestVal

    else :
        bestVal = +INFINITY
        for each child node :
            value = minimax(node, depth+1, true, alpha, beta)
            bestVal = min( bestVal, value)
            beta = min( beta, bestVal)
            if beta <= alpha:
                break
        return bestVal
```

```
// Calling the function for the first time.
minimax(0, 0, true, -INFINITY, +INFINITY)
```

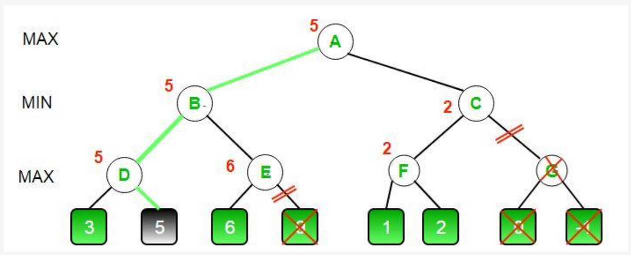


71

Example

```
function minimax(node, depth, isMaximizingPlayer, alpha, beta):

    if node is a leaf node :
        return value of the node

    if isMaximizingPlayer :
        bestVal = -INFINITY
        for each child node :
            value = minimax(node, depth+1, false, alpha, beta)
            bestVal = max( bestVal, value)
            alpha = max( alpha, bestVal)
            if beta <= alpha:
                break
        return bestVal

    else :
        bestVal = +INFINITY
        for each child node :
            value = minimax(node, depth+1, true, alpha, beta)
            bestVal = min( bestVal, value)
            beta = min( beta, bestVal)
            if beta <= alpha:
                break
        return bestVal
```

```
// Calling the function for the first time.
minimax(0, 0, true, -INFINITY, +INFINITY)
```



Next Week

Optimization & Metaheuristics