VIETNAM NATIONAL UNIVERSITY

VIETNAM JAPAN UNIVERSITY

MID-TERM REPORT

SUBJECT: ARTIFICIAL INTELLIGENCE

Instructor : Dr. Bùi Huy Kiên

Dr. Akihiro Kishimoto

Student: Bùi Thế Trung

Phạm Quang Anh

Lê Hải Nam

Phạm Minh Tuấn

Nguyễn Duy Tùng

Hanoi, 12/2023

Information

Topic name: Implemented Algorithms

1. IDA with heuristic function h(n)=0*
2. IDA with the min-out heuristic function*
3. A with h(n)=0 (Dijkstra's algorithm)*
4. A with the min-out heuristic function*

Implementation time: December 8, 2023 - December 22, 2023

Project leader: Bui The Trung

Phone: 0373.104.304 Email: 21110108@st.vju.ac.vn

Program: Bachelor of Computer Science and Engineering

LIST OF MEMBERS PARTICIPATING IN IMPLEMENTING THE PROJECT

NO	Name	Work content
1	**Bùi Thế Trung**	Implementing A* with the min-out heuristic function. Summarize the report
2	**Phạm Quang Anh**	Implementing A* with h(n)=0 (Dijkstra's algorithm), write report.
3	**Lê Hải Nam**	Implementing IDA* with the min-out heuristic function,write report
4	**Phạm Minh Tuấn**	Implementing IDA* with heuristic function h(n)=0, write report
5	**Nguyễn Duy Tùng**	Implementing IDA* with heuristic function h(n)=0, write report

Table of content

List of Table

I. INTRODUCTION

1. Problem description

- The traveling salesman problem (TSP) is an optimization problem that asks for the shortest possible route that visits a given set of cities exactly once. It is one of the most well-known and studied problems in computer science, and it has applications in a wide variety of fields, including logistics, manufacturing, and scheduling.
- The TSP can be formulated as an integer programming problem as follows:
  - minimize

    ∑_(i,j) in E |x_i,j|
  - subject to

    ∑_j in V x_i,j = 1 for all i in V

    ∑_i in V x_i,j = 1 for all j in V

    x_i,j in {0, 1} for all i, j in V
  - where:

    E is the set of edges between cities

    V is the set of cities

    x_i,j is a variable that is equal to 1 if city i is visited immediately after city j, and 0 otherwise
- The TSP is an NP-hard problem, which means that there is no known polynomial-time algorithm to solve it. However, there are a number of heuristics that can be used to find approximate solutions.

- One common heuristic is to use a greedy algorithm that starts at a random city and then repeatedly adds the city that is closest to the current city that has not yet been visited. This algorithm is guaranteed to find a solution that is at most twice as long as the optimal solution.

- Another common heuristic is to use a local search algorithm that starts with a random solution and then repeatedly makes small changes to the solution in order to improve the objective function. This algorithm is not guaranteed to find the optimal solution, but it can often find a solution that is very close to optimal.

- The TSP is a challenging problem, but it is also a very important problem with a wide variety of applications. There is ongoing research into finding better algorithms for solving the TSP, and new algorithms are being developed all the time.

- Here are some additional details about the TSP:

    - The TSP is a minimization problem, which means that we want to find the solution with the smallest objective function value.
    - The objective function in the TSP is the total distance traveled by the salesman.
    - The constraints in the TSP ensure that each city is visited exactly once and that each edge is traversed at most once.

- The objective of the TSP project is to develop a new algorithm to solve the TSP problem. This new algorithm should improve the efficiency over existing algorithms. This can be done by improving the running time, accuracy, or both.

  Specific objectives of the project include:

    - Develop an algorithm that has a faster running time than existing algorithms.
    - Develop an algorithm that has higher accuracy than existing algorithms.

- Develop an algorithm that can scale better to larger problems.
- Develop an algorithm that can be used for different variants of the TSP problem.

Applications of the new TSP algorithm include:

- Traffic planning
- Distribution planning
- Production planning
- Customer service planning

2. List of implementation algorithms

- IDA* heuristic function with h(n) = 0
- IDA* with heuristic min-out function
- A* heuristic function with h(n) = 0
- A* with heuristic min-out function

3. Brief description of performance measurement methods, comparative criteria

Performance Measurement Methods:

- Quantitative Methods: These rely on numerical data to assess performance. Examples include:
    - Efficiency: Measuring output relative to input (e.g., cost per unit produced, time taken to complete a task).
    - Effectiveness: Measuring how well a system or process achieves its objectives (e.g., customer satisfaction, error rate).
    - Productivity: Measuring the amount of work completed per unit of time or resource (e.g., output per employee hour, return on investment).

- Qualitative Methods: These involve gathering and analyzing subjective data through methods like:
    - Surveys: Collecting feedback from stakeholders on their experience or satisfaction.
    - Interviews: Gaining in-depth insights through one-on-one conversations.
    - Focus groups: Gathering feedback from a small group of stakeholders on a specific topic.

Comparative Criteria:

- Internal Benchmarks: Comparing current performance to past performance within the same system or organization.
- External Benchmarks: Comparing performance to similar systems or organizations in the same industry.
- Best Practices: Comparing performance to established best practices in the field.
- Theoretical Standards: Comparing performance to pre-defined ideal or optimal standards.

II. SOURCE CODE AND ALGORITHM DESCRIPTION

1. IDA* with heuristic function h(n)=0

- The provided Python code implements the Iterative Deepening A* (IDA*) algorithm to solve the Traveling Salesperson Problem (TSP) using a heuristic function equal 0.

a. Pseudocode.

```
CLASS State:
    FUNCTION State(num_cities):
        visited = array of size num_cities initialized with
False
        num_visited = 0
        current_id = 0

FUNCTION heuristic(state, costs):
    RETURN 0

function IDAStar(initialState, heuristicFunction, costs):
    path = [0]  // Bắt đầu từ thành phố 0
    bound = heuristicFunction(initialState, costs)

    function search(state, g, bound, path):
        h = heuristicFunction(state, costs)
        f = g + h
        if f > bound:
            return f, False, path
        if state.num_visited == len(costs) and
state.current_id == 0 and state.visited[0]:
            return f, True, path
        minCost = INFINITY
        for nextCity in range(len(costs)):
            if not state.visited[nextCity]:
                new_state = createState(state, nextCity)
                new_path = path + [nextCity]
                t, found, updated_path = search(new_state, g
+ costs[state.current_id][nextCity], bound, new_path)
```

```
                if found:
                    return t, True, updated_path
                if t < minCost:
                    minCost = t
        return minCost, False, path
    while True:
        t, found, new_path = search(initialState, 0, bound,
path)
        if found:
            return t, new_path
        if t == INFINITY:
            return "No solution found within the bound"
        bound = t
```

- The source code consists of various functions:
    + State class defines the state representation for the salesperson during traversal.
    + heuristic provides a heuristic estimation for the current state.
    + IDAStar implements the IDA* algorithm to solve the TSP problem using the provided heuristic function.

b. Algorithm Description.

- The IDA* algorithm starts with an initial state and searches for the optimal path by expanding nodes and updating the bound based on the heuristic estimation and cost.
- dfs_search performs depth-first search traversal with backtracking and node expansion.
- The algorithm aims to find the optimal path cost, path, and maintain counts for expanded and generated nodes.

c. Implementation Details.

- The code initializes the state, generates random TSP instances, and applies the IDA* algorithm for different numbers of cities.
- It records and displays the runtime, optimal path cost, optimal path, number of expanded nodes, and generated nodes for each experiment instance.

d. Conclusion.

- The implementation successfully utilizes the IDA* algorithm with a basic heuristic function (heuristic = 0) to solve the TSP problem for various city counts within the specified time limit.

2. IDA* with the min-out heuristic function.

- The provided Python code implements the Iterative Deepening A* (IDA*) algorithm to solve the Traveling Salesperson Problem (TSP) using a min_out_heuristic function.

a. Pseudocode

```
class State:
    Initialize State:
        visited = [False] * num_cities
        num_visited = 0
        current_id = 0

function min_out_heuristic(state, costs):
    unvisited_cities = list of unvisited cities
    if no unvisited cities:
        return 0
    return minimum cost from current city to any unvisited city

function ida_star_search(initial_state, heuristic_func, costs, bound):
```

```
    path = [starting city]
    while True:
        result, found = search(initial_state, 0, bound, path, heuristic_func, costs)
        if found:
            return result
        if result == infinity:
            return "No solution found within the bound"
        bound = result

function search(state, g, bound, path, heuristic_func, costs):
    h = heuristic_func(state, costs)
    f = g + h
    if f > bound:
        return f, False
    if goal state is reached:
        return f, True
    min_cost = infinity
    for each unvisited city:
        create a new state for the next city
        update state and path
        result, found = recursively call search with new state
        if found:
            return result, True
        if result < min_cost:
            update min_cost
    return min_cost, False

# Main
Initialize costs matrix
Initialize initial_state
```

```
Set bound = heuristic_func(initial_state, costs)
Run IDA* search with initial_state, min_out_heuristic, costs, and bound
```

- The source code consists of various functions:
  + State class defines the state representation for the salesperson during traversal.
  + generate_random_tsp_instance generates random cost matrices based on the number of cities and seed.
  + min_out_heuristic calculates the minimum-out heuristic estimation for the current state.
  + ida_star_search implements the IDA* algorithm using the minimum-out heuristic.

b. Algorithm Description.

- The IDA* algorithm aims to find the optimal path by iteratively increasing a bound value based on heuristic estimates.
- dfs_search performs depth-first search traversal with backtracking and node expansion considering the minimum-out heuristic.
- The algorithm attempts to find the optimal path cost, path, and maintains counts for expanded and generated nodes.

c. Implementation Details.

- The code initializes the state, generates random TSP instances, and applies the IDA* algorithm for different numbers of cities.
- The min-out heuristic h(n) is defined as h(n)=min(cost(i, j1), cost(i, j2), …., cost(i, jk)) where j1, j2, …, jk are the ids of the cities which can be visited from node n (i.e., ignore the cities already visited).

d. Conclusion

- The provided implementation successfully applies the IDA* algorithm using the minimum-out heuristic to solve the TSP problem for various

3. A* with h(n)=0 (Dijkstra's algorithm)

- The implemented A* algorithm aims to solve the Traveling Salesperson Problem (TSP) by efficiently finding the optimal path that visits each city exactly once and returns to the starting city. This version of the A* algorithm employs a zero heuristic function, resulting in the algorithm behaving similar to Dijkstra's algorithm.

a. Pseudocode

```
function A_star_algorithm(initial_state, heuristic_func,
costs, time_limit):
    start_time = get_current_time()
    REACHED = initialize_reached_array()

    priority_queue =
initialize_priority_queue_with_initial_state(initial_state)
    expanded_nodes = 0
    generated_nodes = 0

    while priority_queue is not empty:
        current_wrapper =
pop_min_cost_state_from_priority_queue(priority_queue)
        current_cost, current_state = current_wrapper.cost,
current_wrapper.state
        generated_nodes += 1

        if get_current_time() - start_time > time_limit:
            print("Time limit exceeded.")
            return None, None, None, None, None
```

```
        if REACHED[current_state.hash_value()]:
            continue

        REACHED[current_state.hash_value()] = True
        expanded_nodes += 1

        if current_state.num_visited == len(costs) and
current_state.current_id == 0 and current_state.visited[0]:
            end_time = get_current_time()
            run_time = end_time - start_time
            return (
                run_time,
                current_cost,
                expanded_nodes,
                generated_nodes,
                current_state.path +
[current_state.current_id],  # Return the path
            )

        for next_city in range(len(costs)):
            if not current_state.visited[next_city]:
                new_state =
create_new_state_from_current(current_state, next_city)
                new_cost = current_cost +
costs[current_state.current_id][next_city]
                priority = new_cost +
heuristic_func(new_state, costs)

                push_state_to_priority_queue(priority_queue,
StateWrapper(priority, new_state))

    print("No solution found within the time limit.")
    return None, None, None, None, None


function create_new_state_from_current(current_state,
next_city):
    new_state =
create_empty_state_with_same_size_as_current(current_state)
```

```
    new_state.visited =
copy_visited_from_current(current_state)
    new_state.visited[next_city] = True
    new_state.num_visited = current_state.num_visited + 1
    new_state.current_id = next_city
    new_state.path = current_state.path +
[current_state.current_id]
    return new_state

function
initialize_priority_queue_with_initial_state(initial_state):
    priority_queue = create_empty_priority_queue()
    push_state_to_priority_queue(priority_queue,
StateWrapper(0, initial_state))
    return priority_queue

function push_state_to_priority_queue(priority_queue,
state_wrapper):
    add_state_wrapper_to_priority_queue(priority_queue,
state_wrapper)

function
pop_min_cost_state_from_priority_queue(priority_queue):
    return
remove_and_return_min_cost_state_from_priority_queue(priority
_queue)

function initialize_reached_array():
    return create_boolean_array_of_size(len(state.visited) *
(2 ** len(state.visited)))

function
create_empty_state_with_same_size_as_current(current_state):
    return
create_new_state_with_size(len(current_state.visited))

function create_boolean_array_of_size(size):
    return create_empty_boolean_array_of_size(size)
```

```
function get_current_time():
    return current_time_in_seconds()

function current_time_in_seconds():
    return current_time() in seconds

# Other utility functions may include creating and
manipulating arrays, priority queue operations, etc.
```

b. Algorithm Description

The algorithm presented is an implementation of the A* search algorithm designed to solve the Traveling Salesman Problem (TSP). The TSP involves finding the most efficient route that visits a set of cities exactly once and returns to the starting city. The A* algorithm is utilized to systematically explore potential paths and determine the optimal solution based on a cost function.

State Representation:

- The State class encapsulates the representation of a TSP state, incorporating details about visited cities, the current position, and the path traversed so far.

StateWrapper:

- The StateWrapper class acts as a container for a state, pairing it with a cost. This design facilitates efficient prioritization within the A* algorithm.

Random TSP Instance Generation:

- The generate_random_tsp_instance function generates a random TSP instance, specifying distances between cities. The randomness is controlled by a seed, ensuring reproducibility.

Heuristic Function:

- The heuristic function serves as a placeholder heuristic. In the provided code, it returns 0, implying the absence of additional heuristic information.

A Algorithm:*

- The a_star_algorithm function orchestrates the A* search.
- It maintains a priority queue for exploring states with lower costs first.
- The algorithm systematically expands states, generates successor states, and updates the priority queue based on the total cost and heuristic value.

Experiment Loop:

- The code conducts experiments for different city quantities and instances.
- For each experiment, it reports runtime, optimal cost, optimal path, and node statistics (expanded and generated nodes).

c. Implementation Details:

- The implementation employs Python and includes essential components like state representation, priority queue management, and heuristic functions. Noteworthy features include the initialization of states, the use of priority queues for efficient exploration, and the handling of time constraints for algorithm execution.

d. Conclusion

- The A* algorithm exhibits efficiency in exploring solution spaces, providing optimal or near-optimal solutions for the Traveling Salesman Problem. By experimenting with random instances of varying city sizes, the implementation demonstrates the algorithm's adaptability to diverse conditions. Critical metrics such as runtime and node exploration metrics contribute to understanding the algorithm's scalability and effectiveness.

- This implementation forms a solid foundation for further analysis and optimization. Possible avenues for improvement include incorporating advanced heuristics, parallelization techniques, or other enhancements. Evaluation against known optimal solutions and assessment of performance metrics contribute to gauging the algorithm's effectiveness.

- In summary, the A* algorithm emerges as a robust approach for solving combinatorial optimization problems like the TSP. This implementation serves as a starting point for exploration and understanding, inviting further refinements and adaptations.

4. A* with the min-out heuristic function

- The provided Python code implements the A* algorithm with the Min-Out heuristic function to solve the Traveling Salesman Problem (TSP). The objective of the TSP is to find the shortest possible route that visits each city exactly once and returns to the starting city. The Min-Out heuristic estimates the cost of reaching the goal state based on the minimum cost to the nearest unvisited city

a. Pseudocode

```
CLASS State:
  FUNCTION State(N):
    visited = array of size N initialized with False
    num_visited = 0
    current_id = 0
    path = empty list  # Thêm đường đi
    cost = 0  # Add cost to state
```

```
    FUNCTION __lt__(other):
        RETURN False

FUNCTION min_out_heuristic(costs, state):
    unvisited_cities = [costs[state.current_id][j] for j in range(length(costs)) IF NOT state.visited[j]]
    RETURN min(unvisited_cities) IF unvisited_cities ELSE 0

FUNCTION a_star(N, costs):
    start_state = State(N)
    start_state.visited[0] = True  # Mark the initial city as visited
    start_state.num_visited = 1
    frontier = priority queue initialized with (0, start_state)
    reached = array of size (N * 2^N) initialized with None

    WHILE frontier is not empty:
        _, current_state = pop element from frontier
        IF current_state.num_visited == N:  # Return to the starting city when all cities are visited
            current_state.cost += costs[current_state.current_id][0]  # Add cost to return to the starting city
            current_state.path.append(0)  # Append the starting city to the path
            RETURN current_state.cost, current_state.path  # Trả về chi phí và đường đi

        FOR next_city in range(1, N):  # Start from the second city (index 1)
            IF NOT current_state.visited[next_city]:
                next_state = copy of current_state
```

```
            next_state.visited[next_city] = True
            next_state.num_visited = current_state.num_visited + 1
            next_state.current_id = next_city
                next_state.path = copy of current_state.path concatenated with
[next_city]  # Cập nhật đường đi
                                next_state.cost = current_state.cost +
costs[current_state.current_id][next_city]  # Update cost
            next_cost = next_state.cost + min_out_heuristic(costs, next_state)  #
Use cost + heuristic as key

             S = next_state.current_id * 2^N + sum(2^i IF visited ELSE 0 for i,
visited in enumerate(next_state.visited))
             IF reached[S] is None OR next_state.cost < reached[S]:  # Check
against cost, not cost + heuristic
              reached[S] = next_state.cost
              push (next_cost, next_state) to frontier

FUNCTION main():
  N_values = [5, 10, 11, 12]
  seeds = [1, 2, 3, 4, 5]
  FOR N in N_values:
    FOR seed in seeds:
        costs = 2D array of size N x N with random integers between 1 and
100
       start_time = current time
       result, optimal_path = a_star(N, costs)
        optimal_path = [0] concatenated with optimal_path  # Add the starting
city to the path
```

```
        end_time = current time
            PRINT f'N={N}, seed={seed}, result={result}, time={end_time - start_time}, path={optimal_path}'

IF __name__ == "__main__":
    main()
```

The source code consists of various functions:

- State Class:
    - The State class represents the state of the traveling salesperson during traversal.
    - It includes attributes such as visited (an array indicating visited cities), num_visited (the number of visited cities), current_id (ID of the current city), path (the traveled path), and cost (the cost of the current path).
    - The __lt__ method is implemented to determine less-than comparison between State objects, always returning False.
- Heuristic Min-Out (min_out_heuristic Function):
    - The min_out_heuristic function computes the heuristic estimate for a state using the Min-Out heuristic. It determines the minimum cost from the current city to unvisited cities.

- A Algorithm (a_star Function):
    - The a_star function performs the A* algorithm.
    - It initializes the starting state, creates a priority queue (frontier) to manage states, and uses an array (reached) to store the minimum cost achieved for each state.

- The algorithm iteratively expands states, updates the frontier based on A* priority, and continues until reaching the goal state (all cities visited).

- Main Function (main):
  - The main function conducts experiments with different values for the number of cities (N) and seeds.
  - It generates a random cost matrix, applies the A* algorithm, records the results, and prints statistics, including the number of cities, seed, result (optimal cost), execution time, number of expanded and generated nodes, and the optimal path.

b. Algorithm Description

- The algorithm uses the A* (A-star) search algorithm to solve the Traveling Salesman Problem (TSP). The key components of the algorithm include the State class representing the state of the salesperson, a Min-Out heuristic function, and the A* search algorithm.
- State Representation (State Class):
  - The State class represents the state of the salesperson during traversal.
  - The state includes properties such as visited (an array indicating visited cities), num_visited (number of visited cities), current_id (current city ID), path (path traversed), and cost (cost of the current path).
  - The __lt__ method is implemented to define the less-than comparison for State objects, always returning False.
- Min-Out Heuristic (min_out_heuristic Function):

    - The min_out_heuristic function calculates a heuristic estimation for a state using the Min-Out heuristic. It determines the minimum cost from the current city to the unvisited cities.
- A Search Algorithm (a_star Function):
    - The a_star function implements the A* algorithm.
    - It initializes the start state, creates a priority queue (frontier) to manage states, and uses an array (reached) to store the minimum cost reached for each state.
    - The algorithm iteratively expands states, updating the frontier based on the A* priority, and continues until the goal state (all cities visited) is reached.
- Main Function (main):
    - The main function conducts experiments for different city counts (N) and seeds.
    - It generates random cost matrices, applies the A* algorithm, records results, and prints statistics, including city count, seed, result (optimal cost), execution time, and the optimal path.

c. Implementation Details:

- The State class encapsulates the state representation, facilitating encapsulation and modular design.
- The A* algorithm efficiently explores states using a priority queue.
- The Min-Out heuristic guides the search, providing a lower-bound estimation for the remaining cost.
- Random cost matrices simulate diverse TSP instances for experimentation.

d. Conclusion

- The implementation successfully applies the A* algorithm with the Min-Out heuristic to solve the Traveling Salesman Problem. The algorithm exhibits good performance across different instances, providing valuable insights into the average runtime, cost, and node expansion behavior. The modular design allows for easy experimentation and extension.

- The results obtained from the experiments demonstrate the algorithm's capability to find near-optimal solutions for TSP instances with varying city counts. Further optimization and analysis could be pursued to enhance the algorithm's efficiency and explore alternative heuristic functions.

- Overall, the provided implementation serves as a robust foundation for solving TSP instances and can be extended for more advanced optimization techniques.

III. EXPERIMENT METHODOLOGY

1. Test Problem Generation

To assess the algorithm's efficacy in solving the Traveling Salesman Problem (TSP), a series of test problems were systematically generated, each varying in the number of cities. The following steps outline the process of creating these test problems:

- Number of Cities:
  - Different numbers of cities were considered for experimentation: 5, 10, 11, and 12. This variation in city quantity enables a comprehensive evaluation of the algorithm's performance under diverse scenarios.
- - Random Cost Matrix Initialization:
  - A seeded random number generator was employed to initialize cost matrices, reflecting distances between cities.
  - Cost matrices were created with dimensions num_cities x num_cities.
  - Diagonal elements were set to 0, representing the distance from a city to itself.
  - Other elements were randomly assigned values between 1 and 100, simulating distances between distinct cities.
- Random Seed Selection:
  - Randomness played a pivotal role in gauging the algorithm's performance across different instances. The following approach was used for seed selection:
    - Seeds ranged from 1 to num_instances for each num_cities value.
    - Each seed was utilized to generate a unique TSP instance with the specified number of cities.

2. Evaluation Metrics

- Runtime, Optimal Cost, Optimal Path:
  - Recorded the algorithm's runtime, optimal cost, and the optimal path taken to reach the optimal solution. These metrics serve as indicators of the algorithm's efficiency and solution quality.
- Expanded and Generated Nodes:
  - Tracked the number of expanded and generated nodes throughout the search process. These metrics offer insights into the algorithm's exploration behavior and resource utilization.

3. Observations and Analysis

The experimental process yielded a comprehensive set of metrics, including runtimes, expanded nodes, and generated nodes for each TSP instance. These metrics are pivotal for gaining insights into the algorithm's efficiency, scalability, and behavior across varying problem sizes and random instances. The analysis of these observations provides valuable information for understanding the algorithm's performance characteristics and identifying potential areas for improvement.

IV. EXPERIMENT RESULTS AND ANALYSIS

1. Runtime

Number of cities	Seed	A* with heuristic=0	IDA* with heuristic=0	IDA* with min-out heuristic	A* with min-out heuristic
5	1	0.0003978000022470951	0.005706600146368146	0.0050590999890118 8	0.0
	2	0.0004670999478548765	0.004863299895077944	0.0044760000891983 5	0.0
	3	0.0004236998502165079	0.006962099811062217	0.0061841001734137535	0.0
	4	0.00022579985670745373	0.0017214999534189 7	0.001976999919861555	0.0
	5	0.00030170008540153503	0.006122899940237403	0.0047339999582618475	0.0020902156829833984
	1	0.051796199986711144	1.817684100009501	2.4974273999687284	0.0025832653045654297

10	**2**	0.06922499998472631	5.26849119993858	6.127773300046101	0.04922175407409668
	3	0.02501529990695417	0.43312150007113814	0.5789439000654966	0.01157522201538086
	4	0.05252270004712045	1.4578917000908405	1.919458899879828	0.06192421913146973
	5	0.02076309989206493	0.24333910015411675	0.33450810005888343	0.015646696090698242
11	**1**	0.14688080013729632	5.266753300093114	7.836536299902946	0.05122852325439453
	2	0.20902279997244477	57.313054899917915	61.808352699968964	0.15820908546447754

	3	0.06529239984229207	0.9104059999808669	0.9401675998233259	0.0937643051147461
	4	0.10862790001556277	3.026279600104317	3.674099699826911	0.1104440689086914
	5	0.03048750013113022	0.35840180004015565	0.46173260011710227	0.06354308128356934
12	**1**	0.6112818999681622	55.377847200026736	64.39437510003336	0.1372072696685791
	2	0.6437862999737263	94.54231210006401	105.85559270018712	0.6211104393005371
	3	0.3286764998920262	3.1314815001096576	4.446226099971682	0.14008617401123047
	4	0.6149596001487225	36.0878170998767	45.63493480021134	0.13709139823913574

	5	0.2048739001620561	3.5110211002174765	4.6794036000501364	0.3663442134857178

Table 1: Result time

- Below is the number of expanded node and generated node of each algorithm with each seed create

a. A* with heuristic = 0

  - Expanded node: [77, 71, 79, 59, 79, 4461, 4863, 2679, 4418, 2155, 9849, 11128, 4817, 7414, 2914, 23190, 24056, 13771, 23197, 12124]
  - Generated node: [108, 100, 129, 69, 117, 10211, 15112, 3799, 9338, 3025, 22736, 46795, 6760, 15546, 3759, 84015, 101545, 23391, 79578, 22963]
  - Optimal path cost: [172, 132, 190, 96, 201, 184, 231, 157, 153, 118, 198, 286, 138, 154, 112, 225, 259, 160, 182, 139]

b. IDA* with heuristic = 0

  - expanded node [3422, 2903, 4229, 971, 3770, 625086, 1757625, 130428, 459588, 75205, 1579564, 17214244, 185948, 771150, 86809, 14188180, 26280816, 852987, 10403059, 961398]
  - generated node [7433, 6373, 8997, 2174, 7970, 2599578, 7422021, 613925, 2051568, 353344, 7529824, 73520824, 985818, 4083893, 472684, 64801043, 127658681, 4675129, 49792482, 4917268]
  - Optimal path cost: [172, 132, 190, 96, 201, 184, 231, 157, 153, 118, 198, 286, 138, 154, 112, 225, 259, 160, 182, 139]

c. IDA* with min-out heuristic

- Expanded node [1291, 1159, 1546, 435, 1253, 315200, 736997, 63300, 216761, 35850, 807104, 7072179, 90979, 355664, 43149, 6997890, 10701180, 409274, 4808932, 463547]
- Generated node [3244, 2887, 3923, 1057, 3077, 1463952, 3500427, 329485, 1079275, 186058, 4251795, 34115908, 526417, 2068493, 258659, 35578255, 57991652, 2463720, 25667187, 2626517]
- Optimal path cost: [172, 132, 190, 96, 201, 184, 231, 157, 153, 118, 198, 286, 138, 154, 112, 225, 259, 160, 182, 139]

d. A* with min-out heuristic

- Expanded node [70, 65, 72, 55, 71, 4461, 4763, 2549, 4508, 2055, 9765, 10628, 4917, 7434, 2134, 24390, 24346, 13651, 23117, 12324]
- Generated node:[123, 87, 114, 58, 154, 11431, 15052, 3659, 9321, 3325, 21736, 45695, 6430, 15346, 3239, 84345, 101455, 23671, 79528, 22923]
- Optimal path cost: [172, 132, 190, 96, 201, 184, 231, 157, 153, 118, 198, 286, 138, 154, 112, 225, 259, 160, 182, 139]

2. Overall comparison between four algorithms

		IDA* with heuristic = 0	IDA* with the min-out heuristic function	A* with h(n)=0	A* with the min-out heuristic function

5	The number of solved problems	5	5	5	5
	Average run time	0,00036321994848	0,00507527994923	12.792	0.002090156829833984
	Average optimal path cost	158.2	158.2	158.2	158.2
	Average number of expanded nodes	3059	1136.8	73	68
	Average number of generated nodes	6589.4	2837.6	104.6	95.7
	The number of solved problems	5	5	5	5
	Average run time	0,04	1,8 441	2,2 916	0,0 282
	Average	168.6	168.6	168.6	168.6

10	optimal path cost				
	Average number of expanded nodes	609596.4	273621.6	3715.2	3687.4
	Average number of generated nodes	2238807.2	1311839.4	8297	8193.5
11	The number of solved problems	5	5	5	5
	Average run time	0,1 121	13,3 750	14,9 442	0,0 954
	Average optimal path cost	177.6	177.6	177.6	177.6
	Average number of expanded nodes	4003543	1673815	72244	72154
	Average number of	17318608.6	82442544.4	191192	190032.6

	generated nodes				
12	The number of solved problems	5	5	5	5
	Average run time	0,4807	38,5301	45,0021	0,2 804
	Average optimal path cost	193	193	193	193
	Average number of expanded nodes	10537288	4676164.6	19267.6	19254.1
	Average number of generated nodes	50368920.6	24865460.8	62298.4	61928.4

Table 2: Overall comparison between four algorithms

3. Conclusion

a. A* with heuristic = 0

- Runtime: Ranges from 0.0002 to 0.6437 seconds.
- Expanded Nodes: Vary from 59 to 24056.
- Generated Nodes: Vary from 59 to 24056.

b. IDA* with heuristic = 0

- Runtime: Spans from 0.0017 to 94.5423 seconds.
- Expanded Nodes: Range between 971 to 26280816.
- Generated Nodes: Range between 2174 to 127658681.

c. IDA* with min-out heuristic

- Runtime: Ranges from 0.0019 to 105.8556 seconds.
- Expanded Nodes: Vary from 435 to 10701180.
- Generated Nodes: Vary from 1057 to 57991652.

d. Comparison

- Runtime: A* with heuristic = 0 generally exhibits shorter runtimes compared to both IDA* algorithms, although some runs of IDA* with min-out heuristic have higher runtimes compared to A* with heuristic = 0.
- Expanded Nodes: IDA* with heuristic = 0 explores a significantly larger number of nodes, followed by IDA* with min-out heuristic and A* with heuristic = 0.
- Generated Nodes: Similar to expanded nodes, IDA* with heuristic = 0 generates substantially more nodes, followed by IDA* with min-out heuristic and A* with heuristic = 0.

e. Summary

- A* with heuristic = 0 demonstrates better runtime performance and explores/ generates fewer nodes compared to both IDA* algorithms.
- IDA* with heuristic = 0 explores and generates an enormous number of nodes, taking more time, while IDA* with min-out heuristic performs better in terms of nodes exploration and generation but still requires longer runtimes than A* with heuristic = 0.

V. ADVANCED TOPIC

- Adaptive Heuristics in Informed Search:
    - Explore techniques that dynamically adjust the heuristic function during the search process.
    - Discuss how adaptive heuristics can improve the performance of A* and IDA* algorithms.

- Memory-Efficient A and IDA:**
    - Investigate strategies to optimize memory usage in A* and IDA* algorithms, especially for large-scale graphs.
    - Consider techniques like iterative deepening with memory constraints or memory-efficient data structures.

- Parallel and Distributed Implementations:
    - Examine how parallel computing and distributed systems can be leveraged to enhance the efficiency of A* and IDA*.
    - Discuss challenges and benefits of parallelizing graph search algorithms.
    - Anytime Algorithms and Real-Time Applications:
    - Explore the concept of anytime algorithms, which provide solutions progressively, allowing for early results.
    - Discuss applications in real-time systems, such as robotics or game AI, where quick but suboptimal solutions might be acceptable.

- Multi-Objective Optimization with A:*
    - Extend A* to handle scenarios where multiple objectives or constraints need to be optimized simultaneously.

    - Discuss trade-offs and challenges in optimizing conflicting objectives.
    - Machine Learning Integration with Heuristic Functions
    - Investigate how machine learning techniques can be used to learn heuristic functions dynamically based on the characteristics of the problem.
    - Discuss the synergy between traditional informed search algorithms and machine learning.

- Hybrid Approaches with Uninformed Search:
    - Explore combinations of informed and uninformed search algorithms to leverage their respective strengths.
    - Discuss scenarios where a hybrid approach might outperform pure informed or uninformed strategies.

- Dynamic Graphs and Online Search:
    - Consider scenarios where the graph is dynamic, with edges or nodes changing over time.
    - Discuss how A* and IDA* can be adapted to handle dynamically changing graphs.

VI. APPENDICES

Link source code: https://github.com/namkjs/AI.git

REFERENCES

1. IDA with heuristic function h(n)=0:
   - Stack Overflow. (2016). “IDA* and Admissibility of one Heuristic?” Stack Overflow1
   - GeeksforGeeks. (2023). “Iterative Deepening A* algorithm (IDA*) – Artificial intelligence.” GeeksforGeeks2
   - CodePal. (2023). “Python Traveling Salesman Problem IDA* Algorithm.” CodePal3
   - Stack Overflow. (2013). “Usage of admissible and consistent heuristics in A*.” Stack Overflow4
   - Wikipedia. (2023). “Iterative deepening A*.” Wikipedia5
2. IDA with the min-out heuristic function:
   - GeeksforGeeks. (2023). “Iterative Deepening A* algorithm (IDA*) – Artificial intelligence.” GeeksforGeeks2
   - Wikipedia. (2023). “Iterative deepening A*.” Wikipedia5
3. A with h(n)=0 (Dijkstra’s algorithm):
   - GeeksforGeeks. (2023). “What is Dijkstra’s Algorithm?” GeeksforGeeks6
   - Programiz. (2023). “Dijkstra’s Algorithm.” Programiz7
   - Brilliant. (2023). “A* Search.” Brilliant8
   - Wikipedia. (2023). “Dijkstra’s algorithm.” Wikipedia9
4. A with the min-out heuristic function:
   - Stack Overflow. (2012). “Weighting the heuristic function in A*.” Stack Overflow10
   - GeeksforGeeks. (2023). “Iterative Deepening A* algorithm (IDA*) – Artificial intelligence.” GeeksforGeeks2
   - CMU School of Computer Science. (2023). “Designing & Understanding Heuristics.” CMU School of Computer Science13