

## **מעבדה בבינה מלאכותית**

### **דו"ח מעבדה 3**

**শמות:**

עובדיה חטיב, 608272012

אסיל נחאו, 90522121

### **סביבה ההרצה והכליים בהם נעשה שימוש**

- המפונח שהשתמשנו בו: GNU bash 3.2.57
- גרסת הפיתון: 3.9.6
- חבילות לא סטנדרטיות שהשתמשנו בהן: matplotlib 3.9.4

### **הרכבת קובץ הפיתון**

קוד הפיתון רץ באמצעות הפקודה הבאה:

```
python lab3.py <time_limit> <problem> <file_path>
```

```
python lab3.py <time_limit> <problem>
```

כאשר:

- time\_limit: זמן ההרצה המקסימלי. במקרה שזמן ההרצה מגיע לסוף הזמן ההרצה מפסיק והגינום של הפרט בעל הפיתון הטוב ביותר שיוטר שהתקבל עד אז יודפס.
- problem: סוג הבעיה מבין שתי בעיות בהם הקוד שלנו מטפל - CVRP לבעיית ניתוב הרכבים, ו- ACKLEY לבעיית פונקציית Ackley.
- file\_path: הנתיב לקובץ הקלט של הבעיה. ארגומנט זה נדרש רק עבור בעיית ניתוב הרכבים.

דוגמאות לפקודות הריצה חוקיות:

```
python lab3.py 300 CVRP P-n16-k8.vrp.txt
```

```
python lab3.py 500 ACKLEY
```

הערה: הקוד מרים את האלגוריתמים השונים אחד אחרי השני, תוך שהוא מציג את התוצאות לכל אחד, ובסיום מציג את התוצאות של ההשוואה בין האלגוריתמים השונים. על מנת להריץ אלגוריתם אחד בלבד מבין האלגוריתמים נא לבחור אותו כערך של המשתנה הגלובלי (ALGORITHM) המופיע בתחילת הקוד, בנוסף, להחליפן בין שתי השירותות האחרונות בקוד, קר ש- *run\_full\_comparison(input\_file)* וتبוטל הערה המשוררת *.main(input\_file)*.

## הרכבת קובץ ה- EXE

קוד ה- EXE רץ באמצעות שתי הפקודות הבאות:

```
lab3.exe <time_limit> <problem_type> <file_path>
```

כאשר הארגומנטים שהוא מקבל הם אותם ארגומנטים שהסבירו לגבי הריצת קובץ הפיתון.

## יצוג הפרטימ, האוכלוסייה, והאלגוריתמים:

- **בעית ה- CVRP:** הפרט (פתרונות) מיוצג ע"י מופיע של המחלקה *CVRPIndividual* המכיליה 3 שדות: מסלולים, היפותנס(העלות), והאוכלוסייה (אוסף הפתרונות) אליה משתיך הפתרון. שדה המסלולים הוא רשימה המכיליה רשימות המיצגות מסלולים ומכלילה את האינדקסים של הערים במסלול לפי סדר הביקור בהם. בנוסף המחלקה מכילה פונקציה המחשבת את היפותנס של הפתרון, והוא נקראת במקומות שונים בתוך האלגוריתם. היפותנס מחושב כאורך המסלולים הכלל כאשר כל אחד מהמסלולים מתחילה מהמחсон ו חוזר אליו. המרחק בין שני ערים הוא המרחק האיקלידי בין הקואורדינטות של שתי הערים.  
האוכלוסייה מיוצגת ע"י מופיע של המחלקה *CVRPPopulation*, אשר מכיליה בין היתר את השדות: רשימת הפתרונות, רשימות של היפותנס הטוב ביותר, והיפותנס המוצע בכל איטרציה. בנוסף לכך היא מכילה פרטימ המחולצים מקובץ הקלט כגון תכונות הרכבים ומספרים, קואורדינטות הערים, הביקוש של הערים, מטריצה דו-ממדית השומרת את המרחקים בין הערים ומחושבות בעת אתחול המופיע ומשמשת חישובים שונים במהלך ריצת האלגוריתם.

```

class CVRPIIndividual:
    def __init__(self, routes, population):
        self.routes = routes
        self.fitness = 0
        self.population = population

class CVRPPopulation:
    def __init__(self, filepath):
        self.truck_capacity = None
        self.trucks_count = None
        self.coords = {}
        self.demands = {}
        self.depot = None
        self.optimal_solution = 0
        self._parse_file(filepath)
        self.dist_matrix = self._compute_distance_matrix()
        self.individuals = []
        self.best_fitness = []
        self.avg_fitness = []
        self.variance = []

```

- **בעיית פונקציית Ackley:** הפתרון\הפרט מיוצג ע"י מופיע של המחלקה AckleyIndividual, המכילה שדות של רשימה מקדמי הממדים של הפונקציה, הפיטניס, ומופיע האוכלוסייה אליה משתיר הפתרון. הפיטניס מחושב על פי הנוסחה הנתונה. האוכלוסייה מיוצגת ע"י מופיע של המחלקה AckleyPopulation המכילה בנוסף לרשימות הפתרונות, הפיטניס הטוב ביותר והפיטניס הממוצע בכל איטרציה, את הערכים הנתונים בסעיף של הפרמטרים c,b,a, הממד, והחסמים התחתון והעליון.

```
● ○ ●

class AckleyIndividual:
    def __init__(self, vector, population):
        self.population = population
        self.routes = vector
        self.fitness = 0

class AckleyPopulation:
    def __init__(self):
        self.dim = 10
        self.lower_bound = -32.768
        self.upper_bound = 32.768
        self.a = 20
        self.b = 0.2
        self.c = 2 * math.pi
        self.individuals = []
        self.best_fitness = []
        self.avg_fitness = []
        self.variance = []
```

בנוסף כל אחד מהאלגוריתמים מיוצג ע"י מחלקה נפרדת. המכילה בתוכה שדה לאוכלוסייה עליה تعمل, פונקציה ראשית הנקראת `solve` המריצה את האלגוריתם ומדפיסה את הפלט הסופי שלו, ופונקציות עזר אשר נקבעות ע"י `solve`. כמו כן, פונקציות רבות שימוש המשמשות יותר מאלגוריתם אחד, כמו הפונקציה המדפיסה את התוצאה הסופית, נמצאות מחוץ למחלקות.

ערכים הפרמטרים הייחודיים של האלגוריתמים השונים דוגמת הטמפרטורה ההתחלתית, אחוז ההתאדות (evaporation rate), מספר האדים וכו' נקבעו ע"י המכנים הבא: עבור כל ערך אפשרי, האלגוריתם הורץ 5 פעמים על 5 אוכלוסיות ההתחלתיות שונות תוך שימוש בערך זהה, והערך שננתן את התוצאות הכל טובות מבחינת הפיטניס הטוב ביותר/מוצע הפיטניסים הטוב ביותר ביותר הוא שנבחר. הרבה פעמים שני פרמטרים נבחנו ביחד (כמו אחוז ההגירה ומרוחך ההגירה באלגוריתם הגנטי) וההשוואה הייתה בין הקומבינציות של זוגות הערכים של שני הפרמטרים. בחלוקת מהפעמים התוצאות היו שונות עבור שתי הביעות ולכל בעיה אומצה האופציה הטובה בשבילה. כמו כן, לפעמים האופציה שננתנה את הפיטניס הטוב ביותר הייתה שונה בהתאם למוצע הפיטניסים הטוב ביותר ועוד והוא צורך

לבחירה אחת מהם, לא הייתה לנו דרך ספציפית להתמודדות עם מקרים כאלה ובחירה אחד מבין השניים הייתה תלויות סיטואציה. בפרקים שונים בהמשך אנחנו מציגים דיאגרמת עמודות של התוצאות שהתקבלו עבור ערכי הפרמטרים הרלוונטיים. יש לציין שההרצות הנוגעות להשוואות נועשו תוך שימוש בקובץ 10-k-80-A מפני שלדעתנו הוא בגודל ובמורכבות שמאפשרות הבחנה בהבדלים.

מלבד האלגוריתמים Multi-Stage Heuristics ו- Branch and Bound, כל האלגוריתמים רצים עד שאחד משולשת התנאים מתקיים: התוכנות לאופטימום לوكלי, סיום פרק הזמן המקסימלי הניתן לאלגוריתם ארגומנט, הגיעו למספר האיטרציות המקסימלי. כמו כן, מלבד ה- ACO שהוא חלק מה- SLS, העובד אף ורק לביעית ניתוב הרכיבים, כל האלגוריתמים תקפים לשתי הביעות, לפעמים עם מימושים שונים כתלות בבעיה, שינויים שיכולים להיות קטנים דוגמת שורה שונה לכל אלגוריתם בהתאם פונקציה ועד שינויים גדולים כגון פונקציה נפרדת לכל אלגוריתם ש谟ממת את הדברים תוך שימוש בלוגיקה שונה.

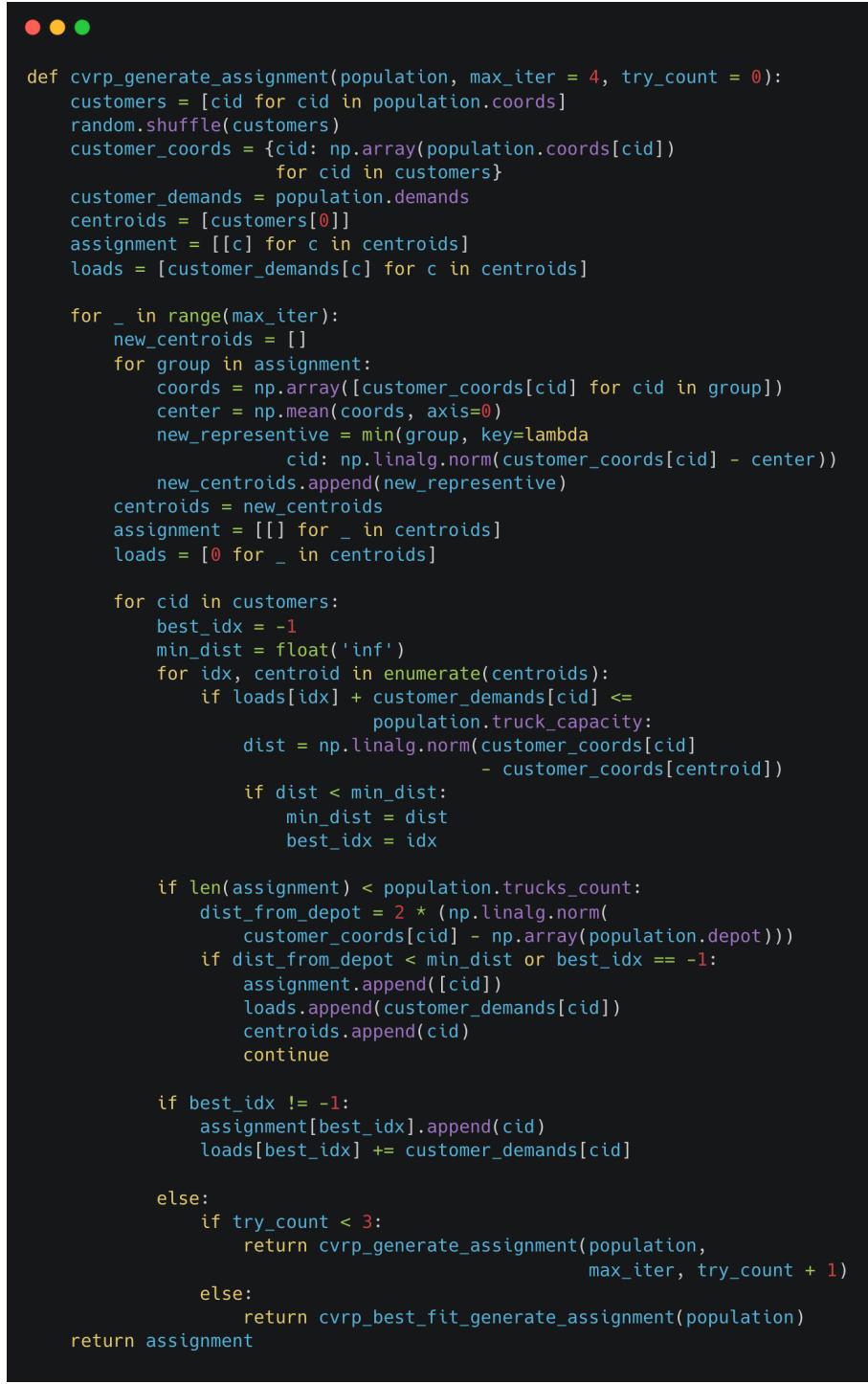
## סעיף 1

האלגוריתם מומש ע"י המחלקה MSHeuristicsAlgorithm הפעלת בשני שלבים עיקריים: **יצירת אוכלוסייה** (אוסף פתרונות) התחלתיים, ואופטימיזציה של הפרטים.

**עבור בעית CVRP**, השלב הראשון מתבצע ע"י השימוש באלגוריתם הבניי על בסיס אלגוריתם K-means אר מותאם לבעה שלנו, כאשר  $K$  הוא מספר הרכיבים בבעיה. הערים באלגוריתם שלנו מחולקים למספר של קבוצות שיכולים להגיע עד  $K$ , כאשר כל קבוצה מייצגת רכב מסלול, ועל סמך כך הערים הנמצאים באותה קבוצה נכללים באותו מסלול. האלגוריתם משייר את העיר לקבוצה מסוימת על בסיס המרחק שלה ממרכז הקבוצה תוך הלקיחה בחשבון של קיבולת הרכב המקסימלית. האלגוריתם מתחילה במסלול אחד המגיע לעיר אחת הנבחרת באופן אקראי, ולכל עיר הוא משייר אותה למסלול קיים או למסלול חדש שבו יהיה את העיר בלבד. ההחלטה על פתיחת מסלול חדש לעיר מסוימת מתקבלת אם המרחק של העיר מהמחסן קטן מכל מרחק ממרכז של אחת הקבוצות (המסלולים) הקיימות אשר הקיבולת שלהם מאפשרת הכנסת הרכב, ובתנאי שמספר המסלול עוד לא הגיע ל-  $K$ .

האתחול מתבסס על ההיויסטיקה שערים קרובות אחת לשניה כדי וחסכו שיחיו באותו מסלול. במקרה שהמדובר עם מצבים קיצוניים בהם הפונקציה לא מצליחה ליצור פרטיים, אם 3 ניסיונות ליצירת פרט נכשלים, פונקציה אחרת המשמשת באלגוריתם First-Fit וממלאת את המסלולים על פי הביקוש שלהם נקראת, הפונקציה מהוות חגורת ביטחון ומטרתה למנוע

מצב בו האלגוריתם לא מצליח לאותחל אוכלוסייה, אולם הפתרונות שהיא מייצרתRndומליים, והתקווה היא שהשלבים הבאים באlgorigithם ישפרו את אותם פתרונות.



```

def cvrp_generate_assignment(population, max_iter = 4, try_count = 0):
    customers = [cid for cid in population.coords]
    random.shuffle(customers)
    customer_coords = {cid: np.array(population.coords[cid])
                       for cid in customers}
    customer_demands = population.demands
    centroids = [customers[0]]
    assignment = [[c] for c in centroids]
    loads = [customer_demands[c] for c in centroids]

    for _ in range(max_iter):
        new_centroids = []
        for group in assignment:
            coords = np.array([customer_coords[cid] for cid in group])
            center = np.mean(coords, axis=0)
            new_representative = min(group, key=lambda
                                      cid: np.linalg.norm(customer_coords[cid] - center))
            new_centroids.append(new_representative)
        centroids = new_centroids
        assignment = [[] for _ in centroids]
        loads = [0 for _ in centroids]

        for cid in customers:
            best_idx = -1
            min_dist = float('inf')
            for idx, centroid in enumerate(centroids):
                if loads[idx] + customer_demands[cid] <=
                   population.truck_capacity:
                    dist = np.linalg.norm(customer_coords[cid]
                                          - customer_coords[centroid])
                    if dist < min_dist:
                        min_dist = dist
                        best_idx = idx

            if len(assignment) < population.trucks_count:
                dist_from_depot = 2 * (np.linalg.norm(
                    customer_coords[cid] - np.array(population.depot)))
                if dist_from_depot < min_dist or best_idx == -1:
                    assignment.append([cid])
                    loads.append(customer_demands[cid])
                    centroids.append(cid)
                    continue

            if best_idx != -1:
                assignment[best_idx].append(cid)
                loads[best_idx] += customer_demands[cid]

        else:
            if try_count < 3:
                return cvrp_generate_assignment(population,
                                                max_iter, try_count + 1)
            else:
                return cvrp_best_fit_generate_assignment(population)
    return assignment

```

איור 1: האלגוריתם המוסף לעליה אשר מתחל קבוצת נתיבים עד  $k$ , כאשר מגד כל קבוצת ערים על סמך המרחק שלהם אחת מהשני.

```
def cvrp_best_fit_generate_assignment(population):
    customers = [cid for cid in population.coords]
    random.shuffle(customers)
    customer_demands = population.demands
    centroids = [customers[0]]
    assignment = [[c] for c in centroids]
    loads = [customer_demands[c] for c in centroids]

    for cid in customers:
        best_idx = -1
        max_load = -float('inf')

        for idx, load in enumerate(loads):
            if load + customer_demands[cid] <= population.truck_capacity:
                if load > max_load:
                    max_load = load
                    best_idx = idx

        if best_idx != -1:
            assignment[best_idx].append(cid)
            loads[best_idx] += customer_demands[cid]
        elif len(assignment) < population.trucks_count:
            assignment.append([cid])
            loads.append(customer_demands[cid])
        else:
            return None

    return assignment
```

איור 2: האלגוריתם השני לייצרת פתרון התחלתי, הוא נקרא רק אם האלגוריתם הראשון נכשל 4 פעמים ברציפות, ועשוה שימוש בעקרון First Fit בקבילת המסלול כקriterיון.

אחרי קביעת הערים המשתתפות בכל מסלול, מטרת השלב השני של האלגוריתם היא לסדר את הערים במסלול בצורה הכי אופטימלית, ועל מנת לעשות זאת הוא משתמש בהיריסטיקה של "הלקוות הקרוב ביותר" כך שבכל שלב הבחירה הקרוב ביותר ללקוות הנוכחי. הלקוות הראשון הוא הכי קרוב למיחס בין הערים באותו מסלול.

```

def tsp_solve(self, individual):
    optimized = []
    total_cost = 0
    for route in individual.routes:
        ordered, cost = self.nn_route_reorder(route)
        optimized.append(ordered)
        total_cost += cost
    individual.routes = optimized
    individual.fitness = total_cost

def nn_route_reorder(self, route):
    if not route:
        return [], 0
    current, cost = min(((node, np.linalg.norm(np.array(
        self.population.coords[node]) - np.array(self.population.depot)))
                           for node in route), key=lambda x: x[1])
    ordered = [current]
    unvisited = set(route)
    unvisited.remove(current)

    while unvisited:
        nearest = None
        min_dist = float('inf')
        for node in unvisited:
            d = self.population.dist_matrix[current][node]
            if d < min_dist:
                min_dist = d
                nearest = node
        cost += min_dist
        ordered.append(nearest)
        current = nearest
        unvisited.remove(nearest)

    cost += np.linalg.norm(np.array(self.population.coords[current]) -
                           - np.array(self.population.depot))
    return ordered, cost

```

איור 3: שתי הֆונקציות האחראיות לשידור הערים בתחום אותו מסלול, תוך כדי חישוב עלות הפתרון (ביחידות מרחק).

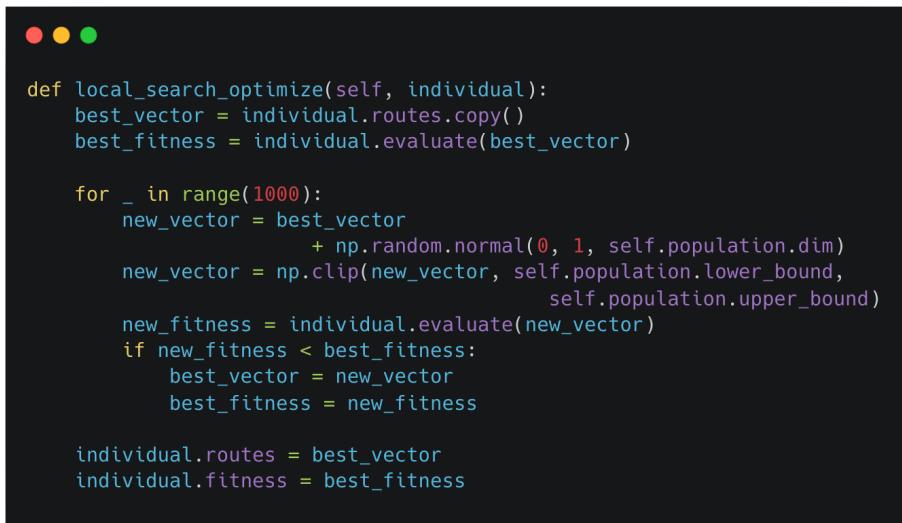
מנגד, **עבור בעיית פונקציית ה- Ackley**, האתחול הוא אקריאי. הוא נבחר להיות צזה בשל התוצאות הטובות שהוא הניב לעומת הדרך האחראית שנבchnerה שדאגה לגיוון וחילקה את המרחב לתתי מרחב כמספר הפרטימ'ופתרונות ובוחרת פרט מכל תת-מרחב באופן רנדומלי. השלב השני שואף לשפר את הפתרונות ולשם כך נעשה שימוש בחיפוש לוקאלי ע"י ייצירת 1000 עותקים של הווקטור ולהוסיף לכל ממד בכל וקטור רעש גאוסיאני, ולבסוף הטוב מבין הווקטורים מחליף את הווקטור הנוכחי.

```

assignment = np.random.uniform(self.population.lower_bound,
                               self.population.upper_bound, self.population.dim)

```

איור 4: הדרך לייצרת פתרון ההתחלתי לבעיית פונקציית Ackley



```

def local_search_optimize(self, individual):
    best_vector = individual.routes.copy()
    best_fitness = individual.evaluate(best_vector)

    for _ in range(1000):
        new_vector = best_vector
        + np.random.normal(0, 1, self.population.dim)
        new_vector = np.clip(new_vector, self.population.lower_bound,
                             self.population.upper_bound)
        new_fitness = individual.evaluate(new_vector)
        if new_fitness < best_fitness:
            best_vector = new_vector
            best_fitness = new_fitness

    individual.routes = best_vector
    individual.fitness = best_fitness

```

איור 5: הfonקציה שמשפרת פתרון בעיית Ackley.

הסביר לגבי ההיררכיות בהם השתמשנו בסעיף, ביחד עם ניתוח הסיבוכיות שלן, נכללים בהסבר לגבי כל ההיררכיות שנעשה בהם שימוש במעבדה בחלק האחרון של הדוח.

## סעיף 2

האלגוריתם ממומש ע"י הfonקציה `ILSAlgorithm`, אשר מתחילה אוכלוסייה (אוסף פתרונות) אותה שיטה שהוסבירה בסעיף הקודם (ע"י האלגוריתם של ה- K-means של בעית CVRP, ובאופן אקראי בקרה של בעית Ackley). האלגוריתם לאחר מכן מוצא בכל איטרציה שכן לכל אחד מהפתרונות, ומאז אותו בקרה שהוא טוב מהנוח', אחרת הוא פועל בהתאם למיניתו היררכית שנבחרה. שיטת ה- CO פועלת באופן שונה ומעלה יוסבר בנפרד בהמשך.

**מציאת השכן מתבצעת** ע"י הfonקציה `find_neighbor` הפעלת תור שימוש בשיטות שונות לכל אחת משתי הבעיות:

בעית CVRP משתמש בשיטות הבאות למציאת שכן:

- opt-2: בוחרת קטע אקראי מຕוך מסלול אקראי והופכת אותו.

```

● ● ●

if method == "2-opt":
    route_idx = random.randint(0, len(routes) - 1)
    route = routes[route_idx]
    if len(route) < 3:
        return individual
    i,j = sorted(random.sample(range(len(route)), 2))
    new_route = route[:i] + list(reversed(route[i:j+1])) + route[j+1:]
    new_routes = individual.routes[:route_idx] +
        [new_route] + individual.routes[route_idx + 1:]
    routes = new_routes

```

- relocate: בוחרת עיר באופן אקראי, מוציאה אותה מהמסלול אליו היא שייכת ומכניסה אותה למסלול אקראי אחר. אם המסלול החדש עדין עומד בגבולת הຕולוה של הרכב היא מחזירה את השcn.

```

● ● ●

elif method == "relocate":
    if len(routes) < 2:
        return individual
    r1 = random.randint(0, len(routes) - 1)
    r2 = random.randint(0, individual.population.trucks_count - 1)
    customer_idx = random.randint(0, len(routes[r1]) - 1)
    customer = routes[r1].pop(customer_idx)
    if r2 >= len(routes):
        routes.append([customer])
        r2 = len(routes) - 1
    else:
        customer_new_idx = random.randint(0, len(routes[r2]))
        routes[r2].insert(customer_new_idx, customer)
    if sum(population.demands[cid] for cid in routes[r2]) \
            > population.truck_capacity:
        return individual
    if not routes[r1]:
        routes.pop(r1)

```

- reposition: בוחרת עיר ומשנה את המיקום שלה באופן אקראי בתוך המסלול אליו היא שייכת.

```

● ● ●

elif method == "reposition":
    route = random.choice(routes)
    if len(route) < 2:
        return individual
    curr_idx, new_idx = random.sample(range(len(route)), 2)
    customer = route.pop(curr_idx)
    if new_idx > curr_idx:
        new_idx -= 1
    route.insert(new_idx, customer)

```

- swap: מחליפה בין שתי ערים באופן אקראי, נבדקת תקינות המסלולים לאחר מכן, ואם הם עומדים בגבולת הרכבים השcn מוחזר.

```

● ● ●

elif method == "swap":
    if len(routes) < 2:
        return individual
    r1, r2 = random.sample(range(len(routes)), 2)
    cus1_idx = random.randint(0, len(routes[r1]) - 1)
    cus2_idx = random.randint(0, len(routes[r2]) - 1)
    routes[r1][cus1_idx], routes[r2][cus2_idx] =
        routes[r2][cus2_idx], routes[r1][cus1_idx]
    if sum(population.demands[cid] for cid in routes[r1])
        > population.truck_capacity or sum(population.demands[cid]
            for cid in routes[r2]) > population.truck_capacity:
        return individual

```

.לוקחת מסלול אקראי ומערבתת את הסדר של הערים בו.

```

● ● ●

elif method == "shuffle":
    routes = [r[:] for r in individual.routes]
    route_idx = random.randint(0, len(routes) - 1)
    random.shuffle(routes[route_idx])

```

עבור בעית Ackley, השימוש היה בשיטות הבאות:

.הוסף רעש, שהוא ערך שבין -0.1 ל- 0.1, לממד אקראי של הווקטור.

```

● ● ●

if method == "shift_one":
    i = random.randint(0, population.dim - 1)
    delta = np.random.uniform(-0.1, 0.1)
    new_vector[i] += delta
    new_vector[i] = np.clip(new_vector[i],
        population.lower_bound, population.upper_bound)

```

.הוסף רעש, וקטור שהערכים שלו בין -0.05 ל- 0.05, לווקטור שנבחר

.באופן אקראי (כל הממדים מושפעים).

```

● ● ●

elif method == "shift_all":
    noise = np.random.uniform(-0.05, 0.05, population.dim)
    new_vector += noise
    new_vector = np.clip(new_vector,
        population.lower_bound, population.upper_bound)

```

.שינויי הערך של אחד הממדים בערך רנדומלי.

```

● ● ●

    elif method == "set_random":
        i = random.randint(0, population.dim - 1)
        new_vector[i] = np.random.uniform(population.lower_bound,
                                         population.upper_bound)

```

במקרה שהמייטה היריסטיקה היא Tabu Search, האלגוריתם מתחילה רשימת טابו בגודל פי 3.5 מגודל האוכלוסייה, ומתחזק אותה במהלך האלגוריתם, כך שהוא מוסיף פתרון חדש לסופה, ומוציא פתרון מהתחלתה במקרה שהוא מלאה. כמו כן, ובשל צרכי יעילות, נעשה שימוש במבנה נתונים מסווג set על מנת למצוא אם פתרון מסוים קיים או שלא והוא מעודכן בהתאם לרשימה, ויחד מרכיבים את טבלת הטابו. לבחירת גודל הטבלה נבחנו כל הגודלים האפשריים בין  $0.25^*$ גודל האוכלוסייה ל- $-8^*$  גודל האוכלוסייה בקפיצות של 0.25. התוצאות במקורה ה-CVRP (שתי הדיאגרמות הראשונות) היו מאד קרובות והבדלים לא היו משמעותיים. מנגד, כן נצפו הבדלים במקרה של בעיית Ackley (שתי הדיאגרמות האחרונות). הערך נבחר להיות  $3.5^*$  גודל האוכלוסייה נבחר בשל היינו מופיע ב- 3 עד 4 הדיאגרמות, ובמיוחד שהוא נתן התוצאות הכי טובות במקרה של בעיית Ackley.

האלגוריתם מחלץ לכל פרט את השכנים שלו באמצעות כל אחת מהשיטות ובוחר מתוכם את הכי טוב בתנאי שהוא לא נמצא בתוך הטבלה.

```

● ● ●

def tabu_hash_initialize(self):
    for ind in self.population.individuals:
        if PROBLEM == "CVRP":
            ind_hash = str(sorted(([tuple(route) for route in ind.routes])))
        elif PROBLEM == "ACKLEY":
            ind_hash = str(np.round(ind.routes, decimals=2).tolist())
        self.tabu_list.append(ind_hash)
        self.tabu_set.add(ind_hash)

```

איור 6: הפונקציה העשויה תחול ל-Tabu Hash Table, נראית בתחלת האלגוריתם ומתחילה טבלה עם אוסף הפתרונות הקיימים (המאוחROL).

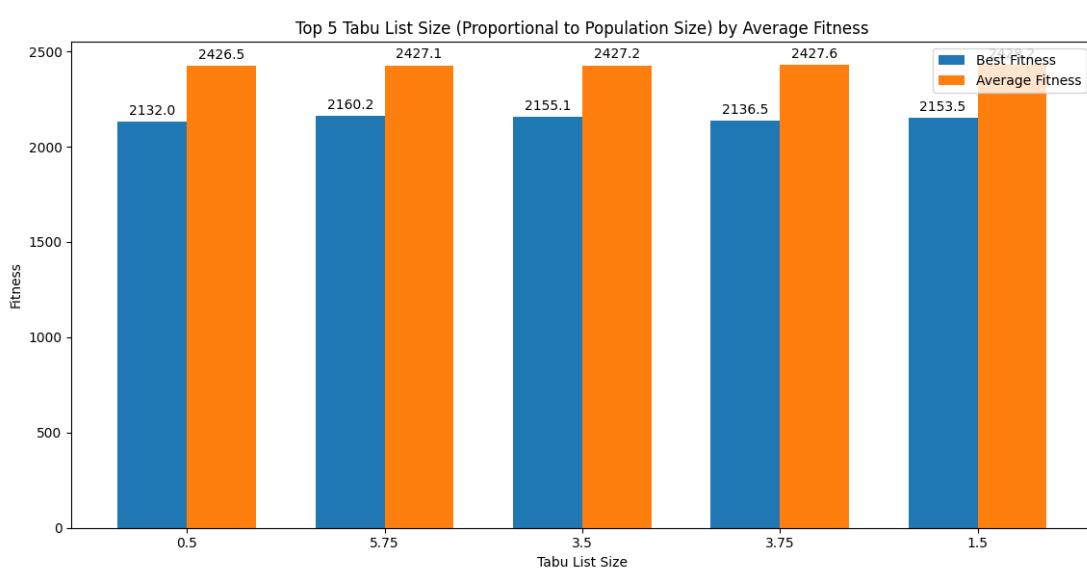
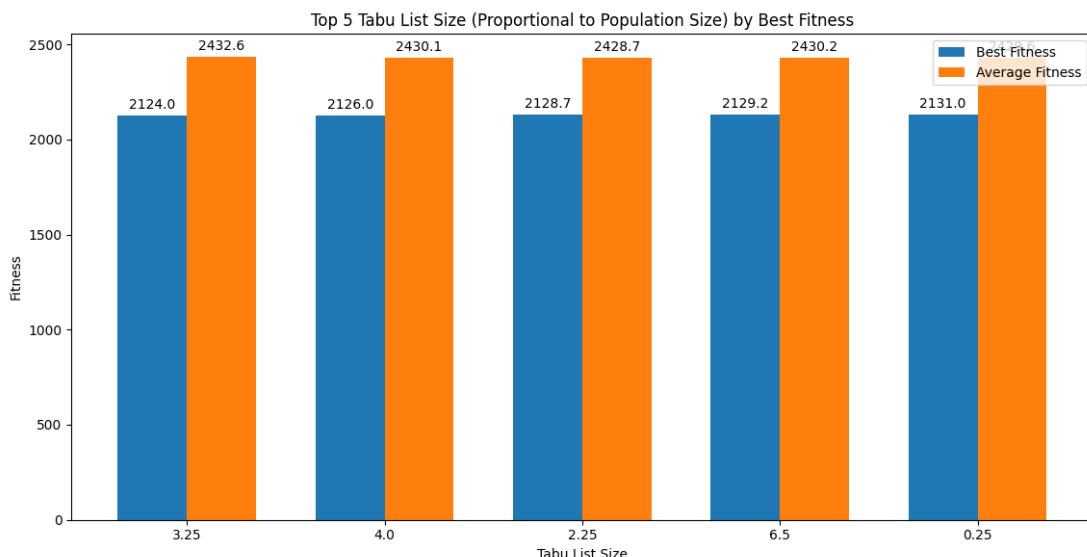


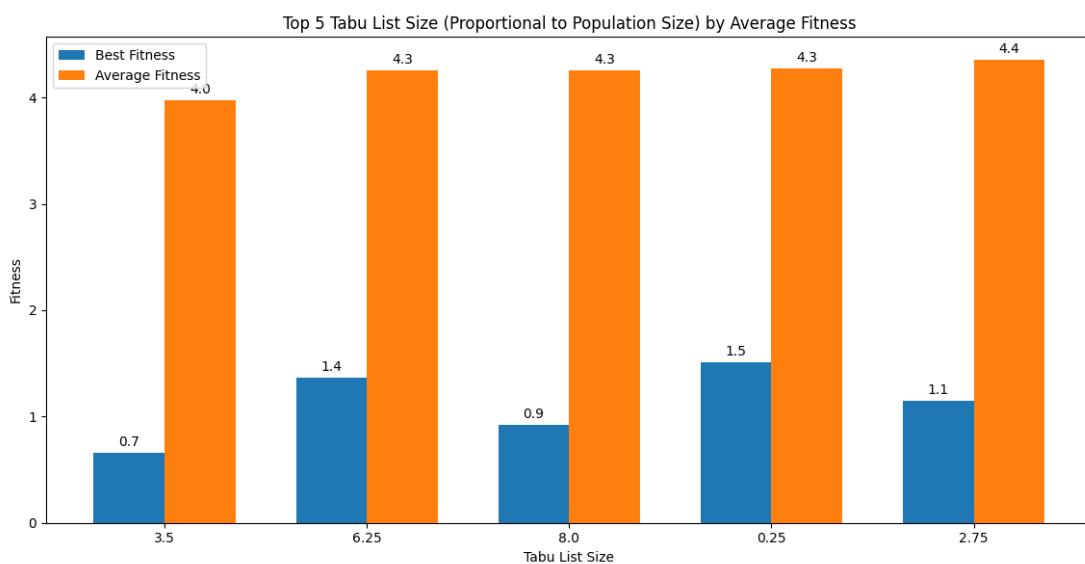
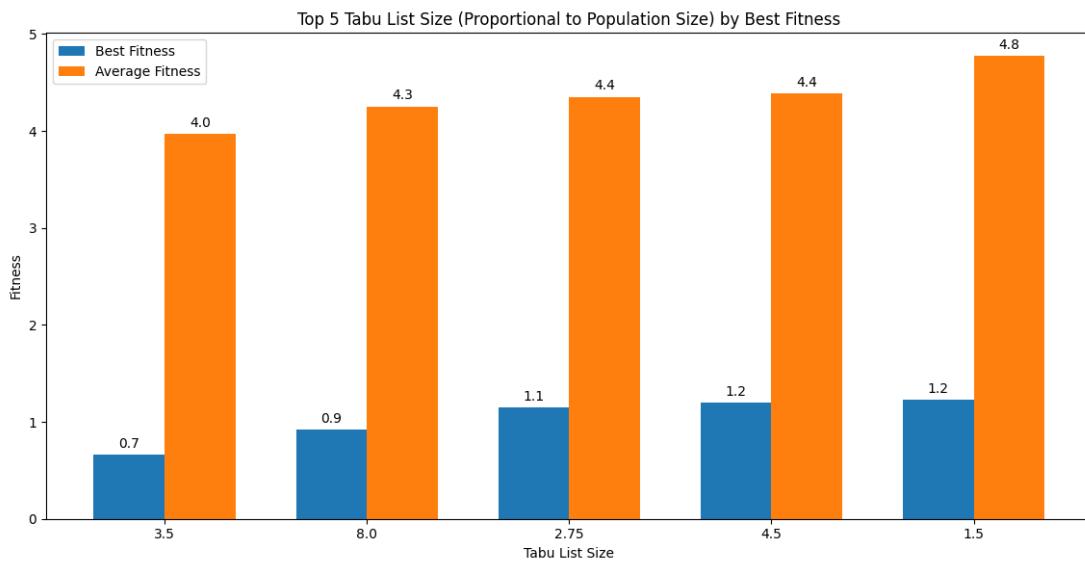
```

if neighbor_hash not in self.tabu_set:
    individual.routes = neighbor.routes
    individual.fitness = neighbor.fitness
    self.tabu_list.append(neighbor_hash)
    self.tabu_set.add(neighbor_hash)
    if len(self.tabu_list) > POPULATION_SIZE * 3.5:
        oldest = self.tabu_list.pop(0)
        self.tabu_set.discard(oldest)

```

איור 7: האחזקה של טבלת ה-*Tabu Hash*, עד שמתבצע עבור כל פתרון שמתווסף.





ה- Simulated Annealing במקום זאת בוחרת שיטת שכנות באופן אקראי ומחלצת את השכן על פיה, אם הוא יותר טוב מהפרט הנוכחי היא מאמצת אותו במקומו, אחרת, היא מאמצת אותו בהסתברות שהשיטה Simulated Annealing קובעת. בסוף כל איטרציה הטמפרטורה מעודכנת, כך שהיא מוכפלת ב- Cooling rate .

```
if neighbor.fitness < individual.fitness:
    individual.routes = neighbor.routes
    individual.fitness = neighbor.fitness
else:
    toReplace = simulated_annealing(individual.fitness,
                                      neighbor.fitness)
    if toReplace:
        individual.routes = neighbor.routes
        individual.fitness = neighbor.fitness
```

איור 8: הגיון העבודה של האלגוריתם בשימוש ב- *Simulated Annealing*.

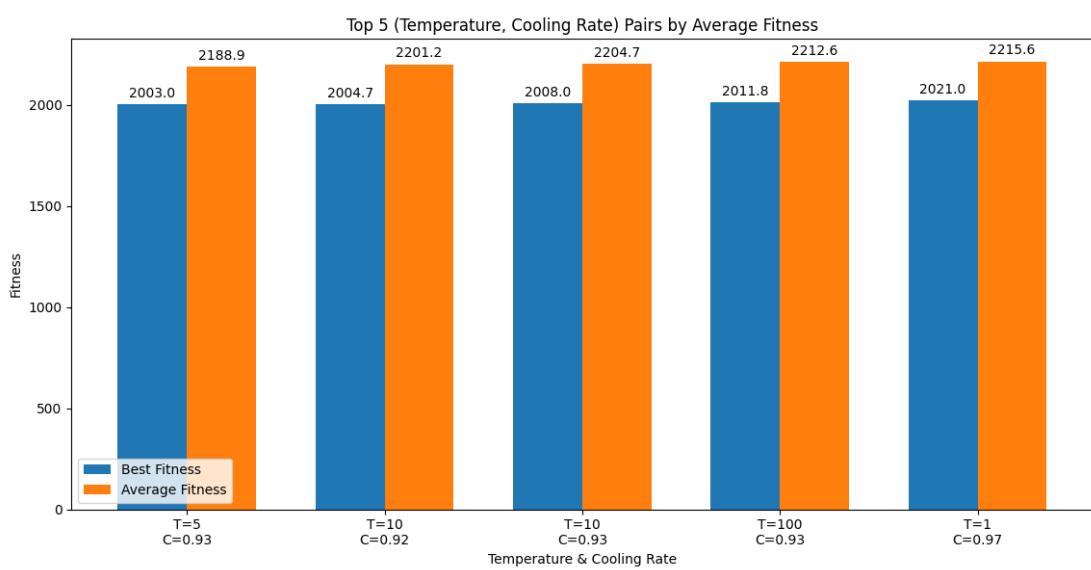
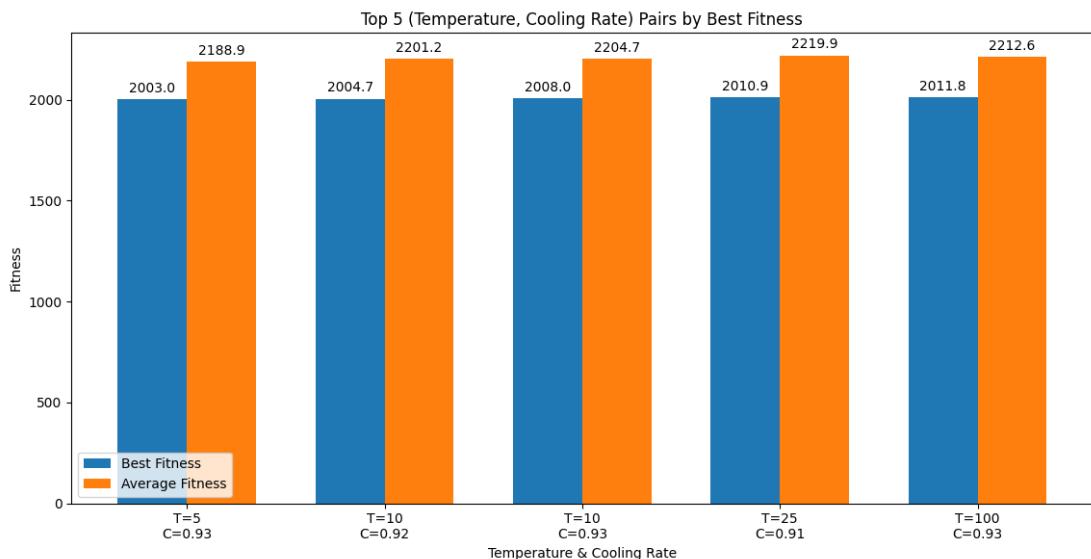
```
def simulated_annealing(individual_fitness, neighbor_fitness):
    delta = neighbor_fitness - individual_fitness
    probability = math.exp(-delta / CURRENT_TEMPERATURE)
    if random.random() < probability:
        return True
    return False
```

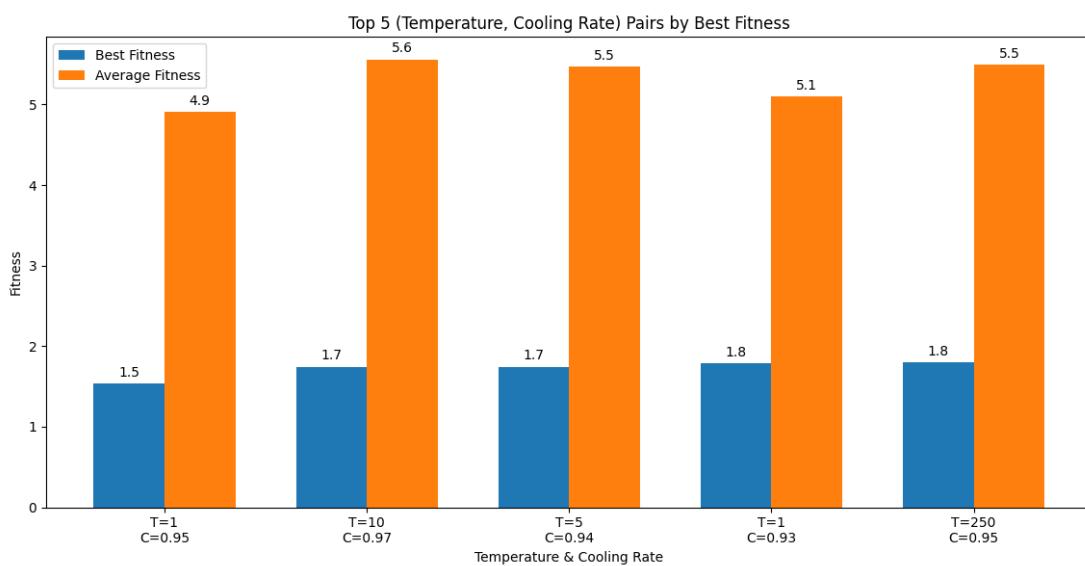
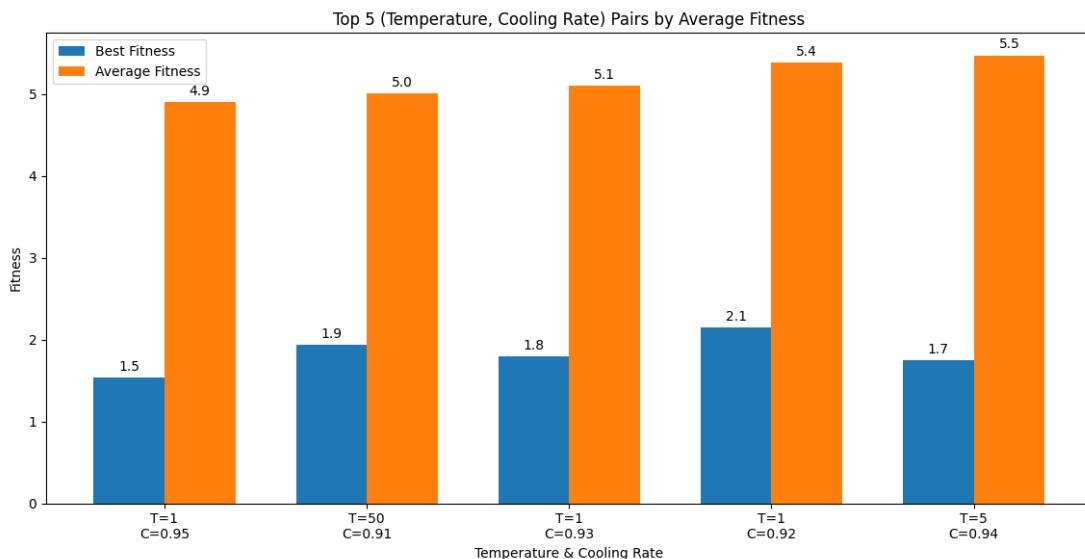
איור 9: פונקציית ה- *True*, אם היא מחזירה *True*, השכן מתתקבל.

```
def update_temperature( ):
    global CURRENT_TEMPERATURE, COOLING_RATE
    CURRENT_TEMPERATURE = COOLING_RATE * CURRENT_TEMPERATURE
```

איור 10: הפונקציה האחראית על עדכון הטמפרטורה בסוף כל איטרציה.

הערכים של ה- *Temperature* ההתחלתי וה- *Cooling rate* של השיטה נבחרו להיות שונים בהתאם לבעיה על סמך תוצאות ההשוואה המצורפות להלן:





היריסטיית Discrete Ant Colony Optimization פועלת באופן שונה. כאשר היא מתחילה את ה- Pheromone בין כל שתי ערים להיות 1.0 וمعدכנת אותם אחרי כל איטרציה. בכל איטרציה היא בונה מסלולים חדשים ע"י בחירת עיר התחלתית במסלול באופן אקראי וכל עיר אחרת מתווספת בהתאם לאם עם הוספה עדין המסלול יעמוד בתנאי התכולה של הרכיב ובאופן יחסית לערך ה- Pheromone שלא מtower ה- Pheromones של שאר הערים העומדים בתנאי בהתאם לנוסחה.

```
def pheromone_initialize(self):
    for i in self.population.coords:
        for j in self.population.coords:
            if i != j:
                self.pheromone[(i, j)] = 1.0
```

איור 11: הפקציה האחראית על אתחול ה- *Pheromone*, נkrאות בתחילת האלגוריתם.

```
def pheromone_update(self):

    for key in self.pheromone.keys():
        self.pheromone[key] *= ACO_EVAPORATION_RATE

    for individual in self.population.individuals:
        for route in individual.routes:
            for i in range(len(route) - 1):
                key = (route[i], route[i + 1])
                if key[0] != key[1]:
                    self.pheromone[key] += ACO_Q
                                / individual.fitness
```

איור 12: הפקציה האחראית על עדכון ה- *Pheromone*, נkrאות בתום כל איטרציה.

```

def aco_solution_construct(self):
    routes = []
    unvisited = set(self.population.coords.keys())
    current = random.choice(list(unvisited))
    load = self.population.demands[current]
    curr_route = [current]
    unvisited.remove(current)
    while unvisited:
        pheromone_prob = []
        for customer in unvisited:
            if load + self.population.demands[customer] <= self.population.truck_capacity:
                tau = self.pheromone.get((current, customer), 1.0)
                eta = 1.0 / (self.population.dist_matrix[current][customer] + 1e-6)
                pheromone = (tau ** ACO_ALPHA) * (eta ** ACO_BETA)
                pheromone_prob.append((customer, pheromone))

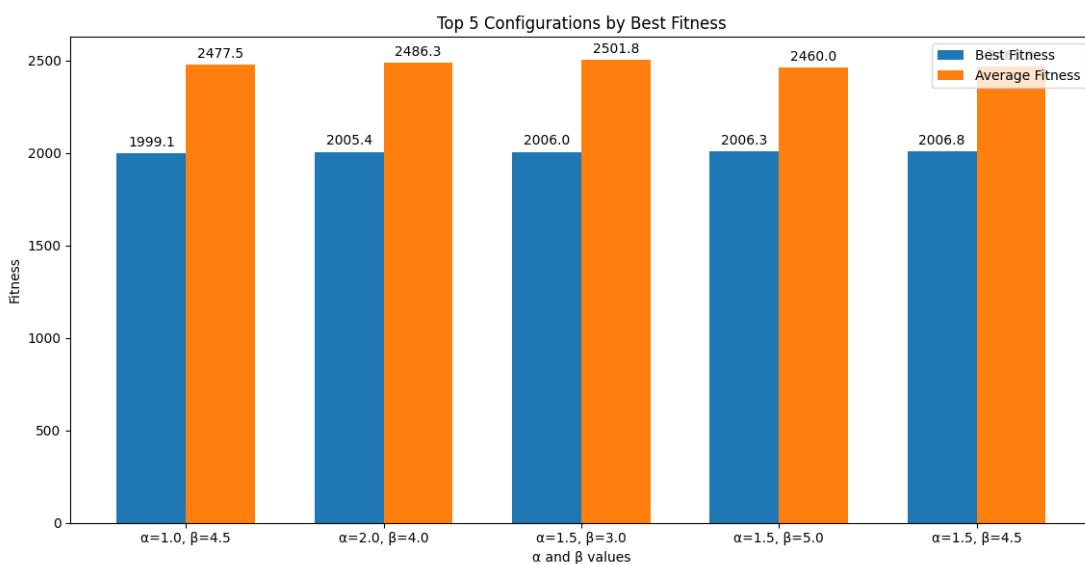
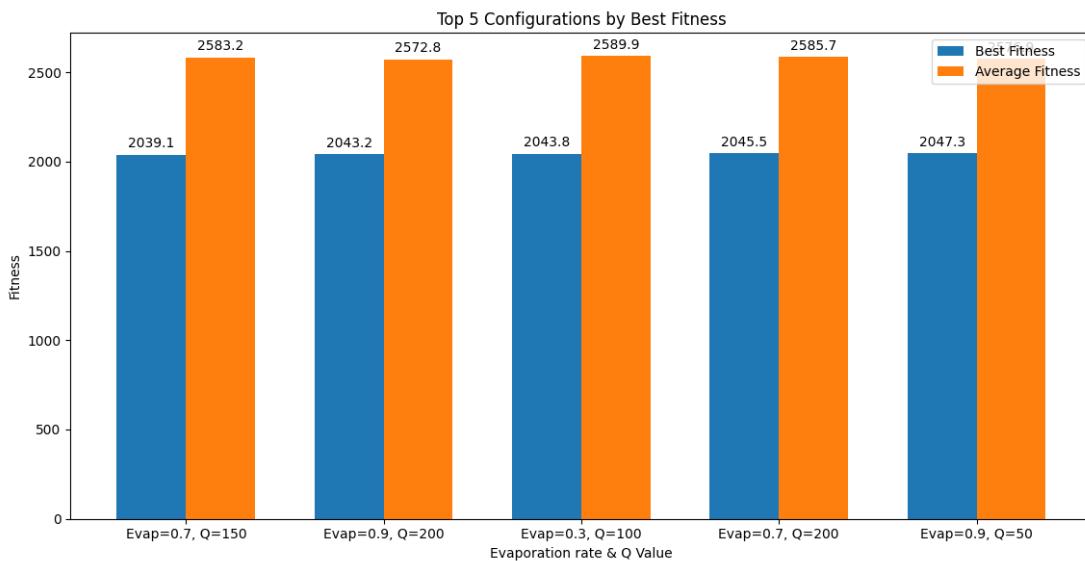
        if pheromone_prob:
            next_customer = None
            pheromone_total = sum(p[1] for p in pheromone_prob)
            if pheromone_total > 0:
                pheromone_prob = [(cust, prob / pheromone_total) for cust, prob in pheromone_prob]
                random_choice = random.random()
                acc = 0
                for cust, prob in pheromone_prob:
                    acc += prob
                    if acc >= random_choice:
                        next_customer = cust
                        break
                curr_route.append(next_customer)
                load += self.population.demands[next_customer]
                unvisited.remove(next_customer)
                current = next_customer
            else:
                routes.append(curr_route)
                if unvisited:
                    current = random.choice(list(unvisited))
                    load = self.population.demands[current]
                    curr_route = [current]
                    unvisited.remove(current)

        if curr_route:
            routes.append(curr_route)
    new_individual = CVRPIndividual(routes, self.population)
    new_individual.evaluate()
    return new_individual

```

איור 13: הפונקציה האחראית על בניית פתרון.

בחירת ערכי הפרמטרים השונים בהם היערטטיקה משתמשת ( $\text{Evaporation rate}$ ,  $Q$ ,  $\alpha$ ,  $\beta$ )  
הו בהתאם להשוויה שנעשתה ותוצאותיה מצורפות להלן:



### סעיף 3

האלגוריתם ממומש באמצעות המחלקה `GAAlgorithm`. היא מתחילה בכך שהיא מתחילה את האוכלוסייה באותה דרך אתחול שהוסבירה מקודם בסעיף א', ומחלקת את האוכלוסייה לאיים שווים בגודלים.

```

def individual_islands_initialize(self):
    individual_idx = 0
    while individual_idx < POPULATION_SIZE:
        if PROBLEM == "CVRP":
            assignment = cvrp_generate_assignment(self.population)
            if assignment is not None:
                new_ind = CVRPIndividual(assignment, self.population)
        elif PROBLEM == "ACKLEY":
            assignment = np.random.uniform(self.population.lower_bound,
                                           self.population.upper_bound, self.population.dim)
            if assignment is not None:
                new_ind = AckleyIndividual(assignment, self.population)
        if assignment is not None:
            new_ind.evaluate()
            island_idx = individual_idx % ISLANDS_COUNT
            self.islands[island_idx].append(new_ind)
            individual_idx += 1

```

בכל שלב ושלב כל אי מתנהג כאוכלוסייה בשל עצמה בכר שהוא שומרת את האליטה לשלב הבא, ויצרת פרטימ חדשים כשילוף של שני הורים, שיכולים לעבור מוטציה בהסתברות שנקבעה להיות 0.25.

```

def evolve_island(self, island):
    next_gen = sorted(island, key=lambda ind:
                      ind.fitness)[:int(ELITISM_RATE * len(island))]
    while len(next_gen) < len(island):
        _, p1 = self.select_parents(island)
        _, p2 = self.select_parents(island)
        child = self.crossover(p1, p2)
        if child is None:
            continue
        if random.random() < MUTATION_RATE:
            child = self.mutate(child)
        if child is None:
            continue
        child = CVRPIndividual(child, self.population) if PROBLEM == "CVRP"
                else AckleyIndividual(child, self.population)
        child.evaluate()
        next_gen.append(child)
    return sorted(next_gen, key=lambda ind: ind.fitness)

```

במרווח של מספר קבוע של דורות מתבצעת הגירה של אחוז מסוים מאוכלוסיית כל אי לאו שבא אחרת. המהגרים נלקחים כך שחצי מהם הם הפרטימ הטוביים ביותר באי שלהם, והחצי الآخر באופן רנדומלי.

```

def migrate(self):
    migrants = []
    for i, island in enumerate(self.islands):
        migrants_num = int(len(island) * MIGRATION_RATE * 0.5)
        self.islands[i].sort(key=lambda ind: ind.fitness)
        island_best_migrants = island[:migrants_num]
        self.islands[i] = self.islands[i][migrants_num:]
        random.shuffle(self.islands[i])
        island_random_migrants = self.islands[i][:migrants_num]
        self.islands[i] = self.islands[i][migrants_num:]
        migrants.append(island_random_migrants + island_best_migrants)

    for i in range(ISLANDS_COUNT):
        for m in migrants[i]:
            target = (i + 1) % ISLANDS_COUNT
            self.islands[target].append(m)

```

רוב הפונקציות לקחוות מהמיושם שלנו במערכות הראשונה והשנייה. עברו בעיתה CVRP ו בשל היותה ניתנת לייצוג ע"י פרמטריזה, בדומה לבעה שהתמודדנו אליה במעבדה הקודמת, שיטות בחירת ההורים, השילוף, והמווציה, כמו גם הערכות של המשתנים הקשורים בשיטות הללו נבחרו להיות כאלה שהראו את התוצאות הכי טובות במעבדה הקודמת. לעומת זאת, Ackley, ובשל היותה לא דומה לביעות שהתמודדנו אותה בעבר היה צריך בלהשווות את הביצועים של הקונפיגרציות השונות על הקלט שלו. בעית displacement, swap, insertion, simple ( Ackley השתמשה באותו שיטות מומציה inversion, inversion, scramble ), אולם השתמשה רק באחת משיטות השילוף של CVRP שהוא Order Crossover. השיטות האחרות לדעתי לא תואמות לשימוש של הbijah. בשל כך הוספנו שתי שיטות שילוף נוספת הייחודיות לbijah Ackley ו הם:

Arithmetic Crossover - השיטה בוחרת ערך α אקראי בתחום (0,1), ויצרת פרט

חדש כצירוף של  $\alpha^*$  ההורה הראשון ו-  $(1-\alpha)^*$  ההורה השני.

```

def arithmetic_crossover(self, p1, p2):
    lower_bound = -32.768
    upper_bound = 32.768
    alpha = random.random()
    child = alpha * p1 + (1 - alpha) * p2
    child = np.clip(child, lower_bound, upper_bound)
    return child

```

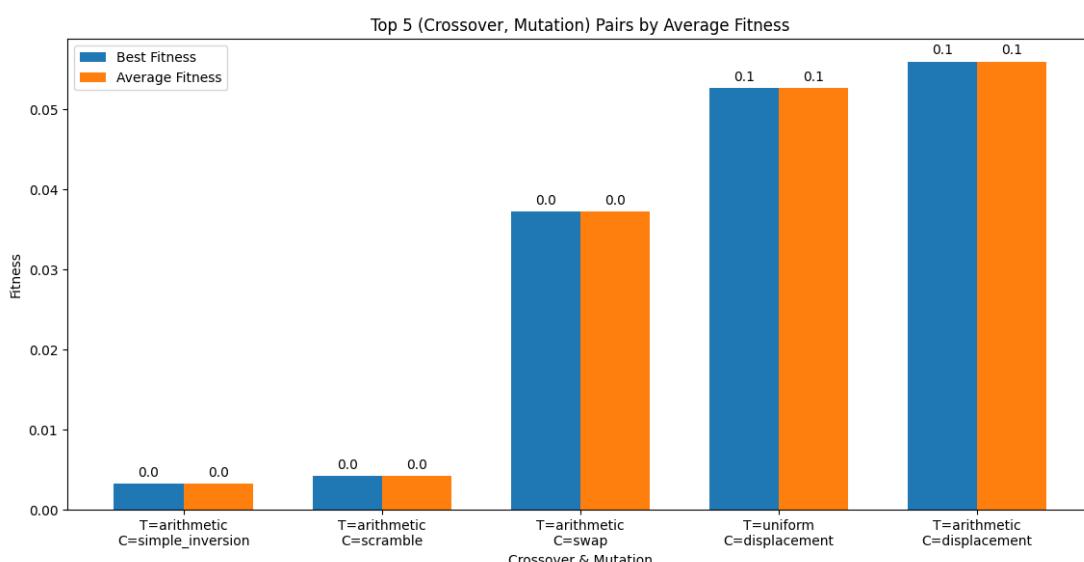
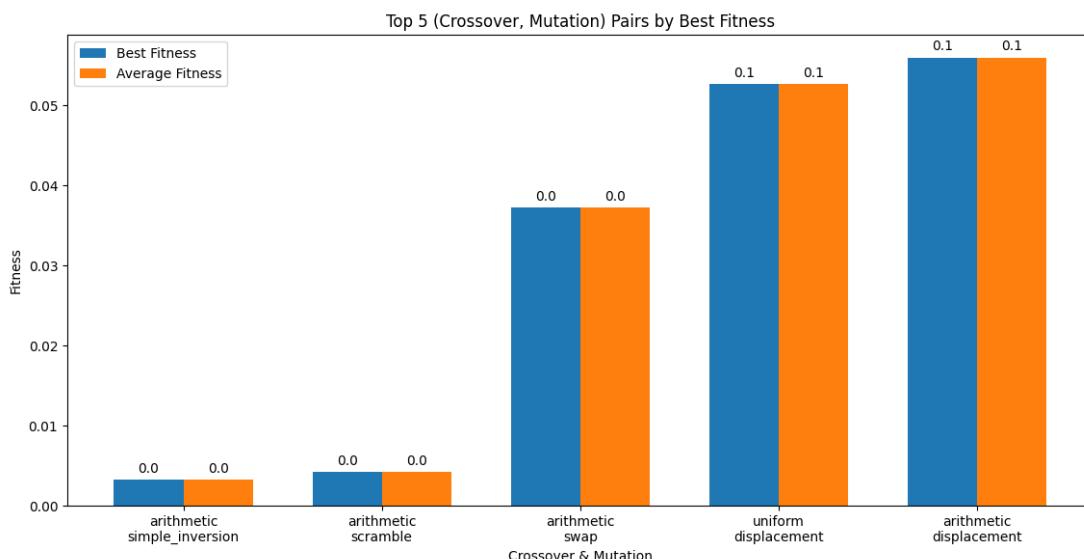
Uniform Crossover - השיטה בונה וקטור חדש כאשר כל ערך של ממד בו לקוח

מאחד ההורים באופן אקראי.



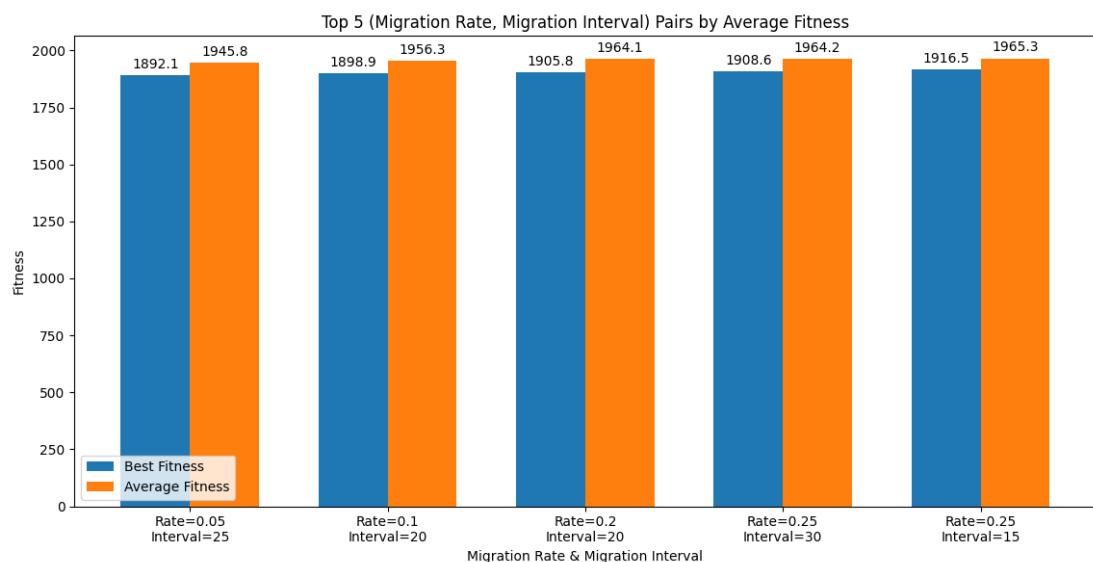
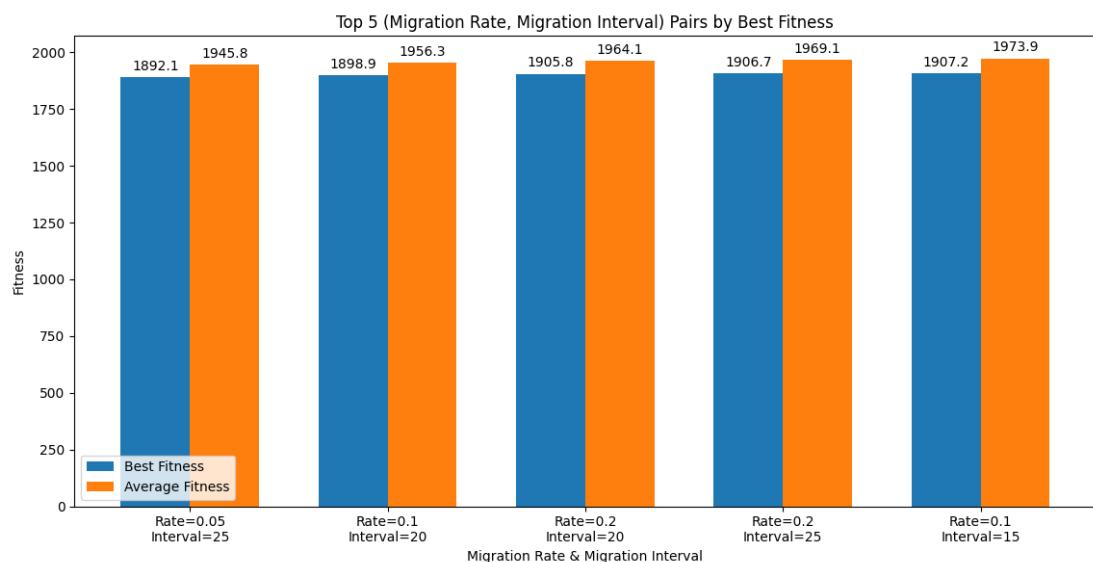
```
def uniform_crossover(self, p1, p2):
    lower_bound = -32.768
    upper_bound = 32.768
    mask = np.random.rand(len(p1)) < 0.5
    child_vector = np.where(mask, p1, p2)
    child = np.clip(child_vector, lower_bound, upper_bound)
    return child
```

18 הזוגות (3 שיטות השילוף ו- 6 שיטות המוטציות) נבחנו עבור בעית Ackley ועל סמך התוצאות נבחרו הזוג (Arithmetic, Simple Inversion) להיות שיטות השילוף והמווציה בהם האלגוריתם משתמש במקרה של הבעיה:

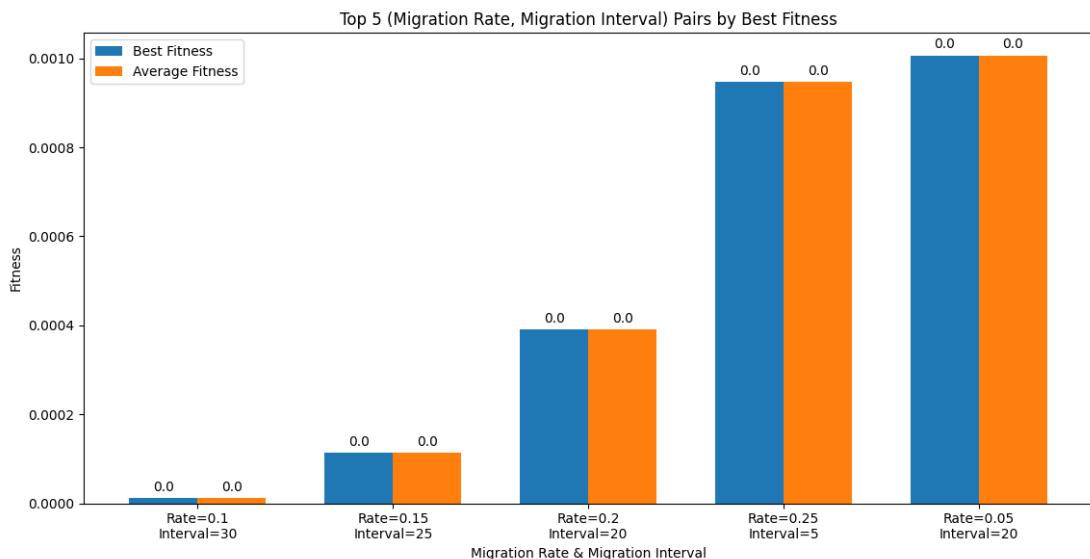


משתנים שהיה חשוב לעשות להם Tuning לשתי הבעיות הם המשתנים הקשורים במודל האיים: מספר האיים, אחוז ההגירה, מרוחק ההגירה, ועל כן היה צריך בלהשווות ביניהם. התוצאות של ההשוואה בין זוגות שונים של **אחוז ההגירה ומרוחק ההגירה** הראו את התוצאות המצורפות למטה עבור כל אחד מהבעיות, כאשר הערכים שנבחנו עבור אחוז ההגירה היו 0.05-0.25 בקפיצות של 0.05, ועבור מרוחק ההגירה 5-30 דורות בקפיצות של 5.

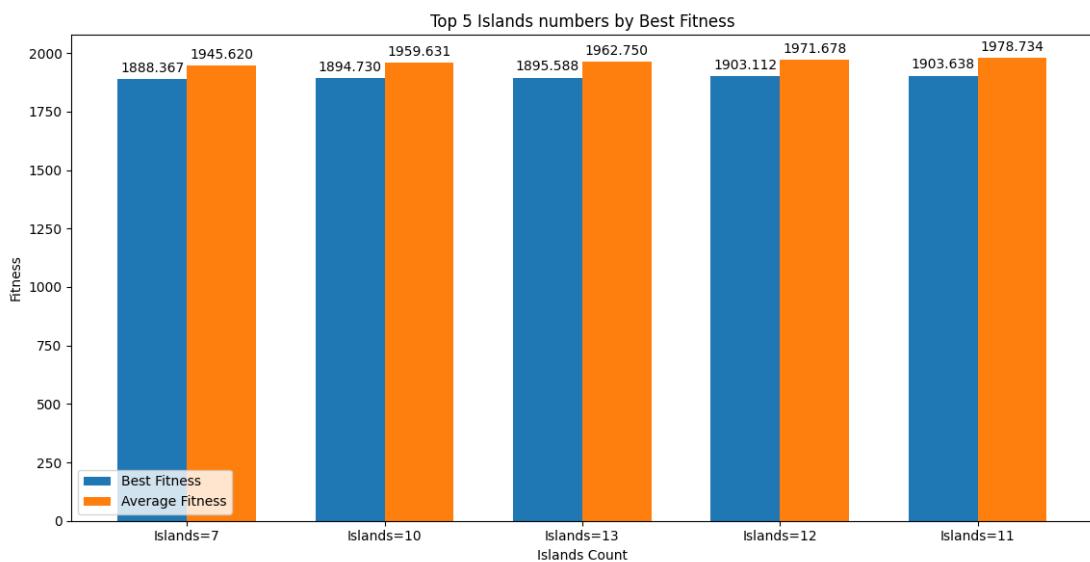
עבור בעיית CVRP:

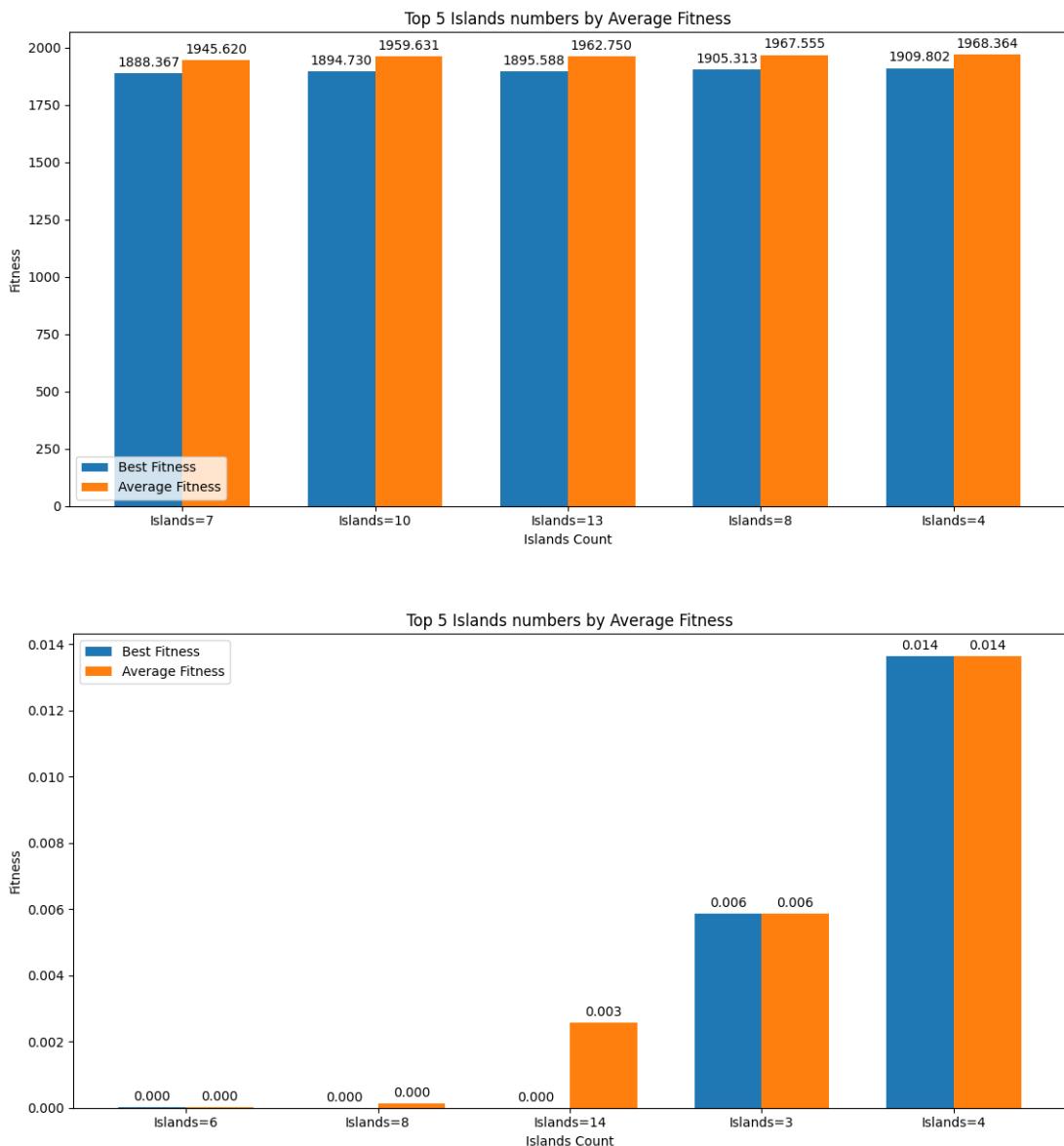


עבור בעית Ackley:



השוואה בין מספר האיים לכמה ביחסו את מספר האיים שבין 2 עד 15 והתוצאות לשתי הבעיות היו כמפורט לעמיה כאשר שתי הראשונות הן בעיית CVRP והאחרונה בעיית Ackley. ובשל כך שבהתאם בזמן מסוים גדול התוצאות היכי טובות מගיעות לפיטנס קטן מדי, שלא מאפשר להשוות בין האופציונות השונות, הגבלנו את הזמן ל-45 שניות ובכך פקטורי הביצוע תחת אילוץ זמן קיבל משקל ממשמעותי בבחירה הערכיהם. על סמך התוצאות קבענו את מספר האיים להיות 7 במקרה שהבעיה היא CVRP ו- 6 אם היא .Ackley





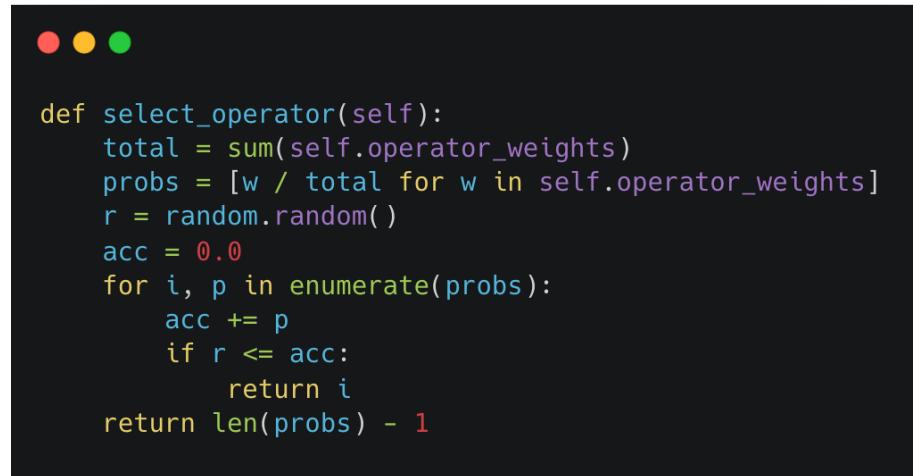
#### סעיף 4

האלגוריתם מוצג ע"י המחלקה `ALNSAlgorithm` שעובדת בצורה הבאה:

1. יצירת אוסף פתרונות ההתחלתית.
2. בכל איטרציה מבוצע הבא:
  - a. לכל פתרון נמצא שיכון לפי אופרטור כלשהו, ונבדק אם הפיטניס שלו קטן מהפיטניס של הפתרון. אם כן, הוא מחליף אותו, אחרת מחליף אותו בהסתברות כלשהי.
  - b. המשקלים של האופרטורים מעודכנים.

אתחול האוכלוסייה מתבצע בהתאם דרך שהוסבירה בסעיף 1 בשיטה שמתבססת על הרעיון של ה- K-means.

בחירה האופרטור היא יחסית למשקל שלו מסה"כ משקל כל האופרטורים:



```
def select_operator(self):
    total = sum(self.operator_weights)
    probs = [w / total for w in self.operator_weights]
    r = random.random()
    acc = 0.0
    for i, p in enumerate(probs):
        acc += p
        if r <= acc:
            return i
    return len(probs) - 1
```

על מנת למצוא שכנים, האלגוריתם משתמש באוטם אופרטורים שהשתמשו בהם בסעיף 2, כאשר עבור בעיית CVRP האופרטורים הם opt-2 ההופכת קטע אקראי הלוקוח ממסלול אקראי, relocate הולכת לлокוח אקראי ומכניסה אותו למקום כלשהו במסלול כלשהו (כולל בדיקת תקינות), reposition הדומה לקודם אך מקומו החדש של הלוקוח יהיה בתוך אותו מסלול, swap המחליפה בין שני לוקוחות (כולל בדיקת תקינות), ו- shuffle הבוחר מסלול אקראי ומעריבב את סדר הלוקוחות בו, ועבור בעיית Ackley האופרטורים הם: shift one, shift all המושיפה "רעש" שהוא ערך אקראי לאחד הממדים, set random המחליפה את הערך של אחד הממדים בערך ערכים אקרים לווקטור, ו- swap המחליפה את הערך של אחד הממדים בכל כלשהו. בחירת האופרטור תהיה באופן הסתברותי ביחס למשקל שלו.

במידה והפיטניס של השכן המתkeletal לא טוב מזה של הפתרון הנוכחי, משתמשים בשיטת Simulated Annealing, שפותחה והוסבירה באlgorigthm LSו בסעיף ב', על מנת להחליט אם לאמץ אותו בכל זאת או שלא.

המשקל של האופרטור מעודכן בכל איטרציה כצירוף של  $0.6 * \text{המשקל הקודם שלו} + 0.4 * \text{ מידת הצלחה שלו באיטרציה הנוכחית}$ . בעיית ניתוב הרכבים, האחוזים משתנים ל- 0.8 ו- 0.2 בהתאם במקרה שהבעיה היא Ackley, כאשר מידת הצלחה מחושבת כמספר הפעםים שבהם השימוש באופרטור הניב שcn בועל פיטניס טוב יותר מחולק במספר הפעמים הכללי בהם נעשה שימוש באופרטור.

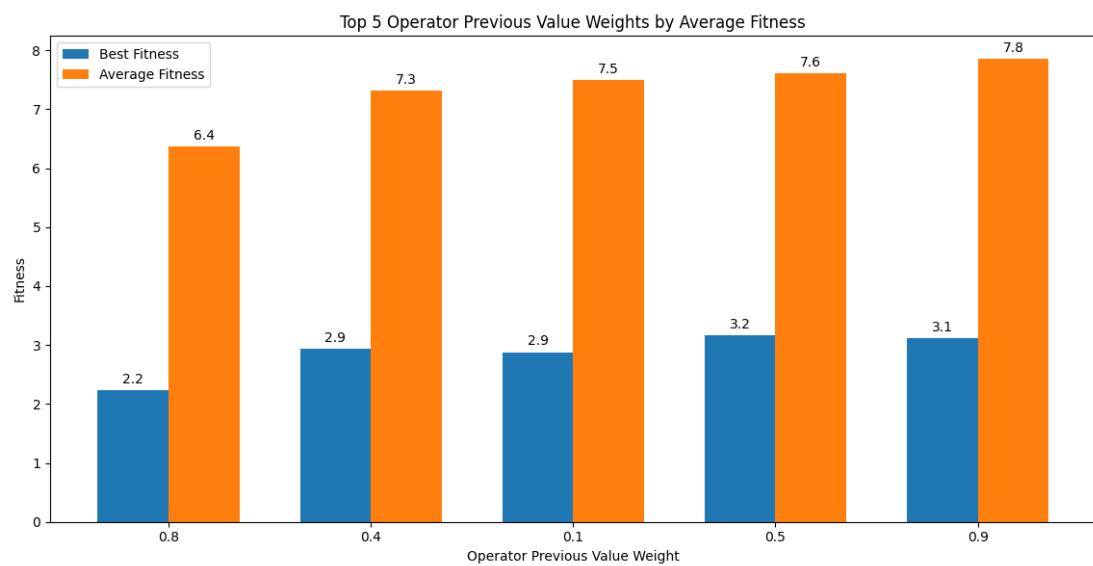
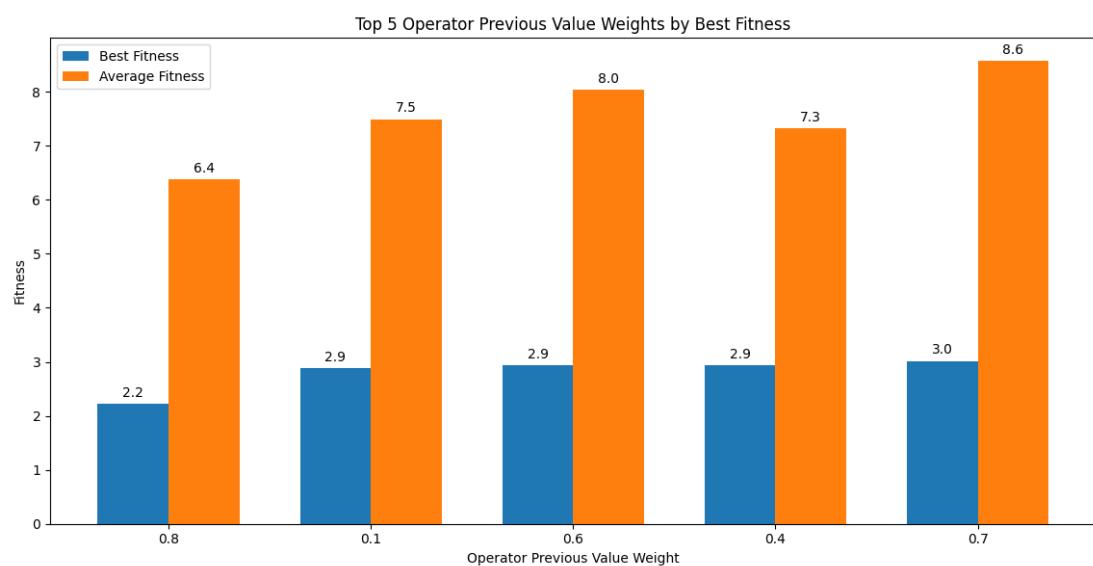
```

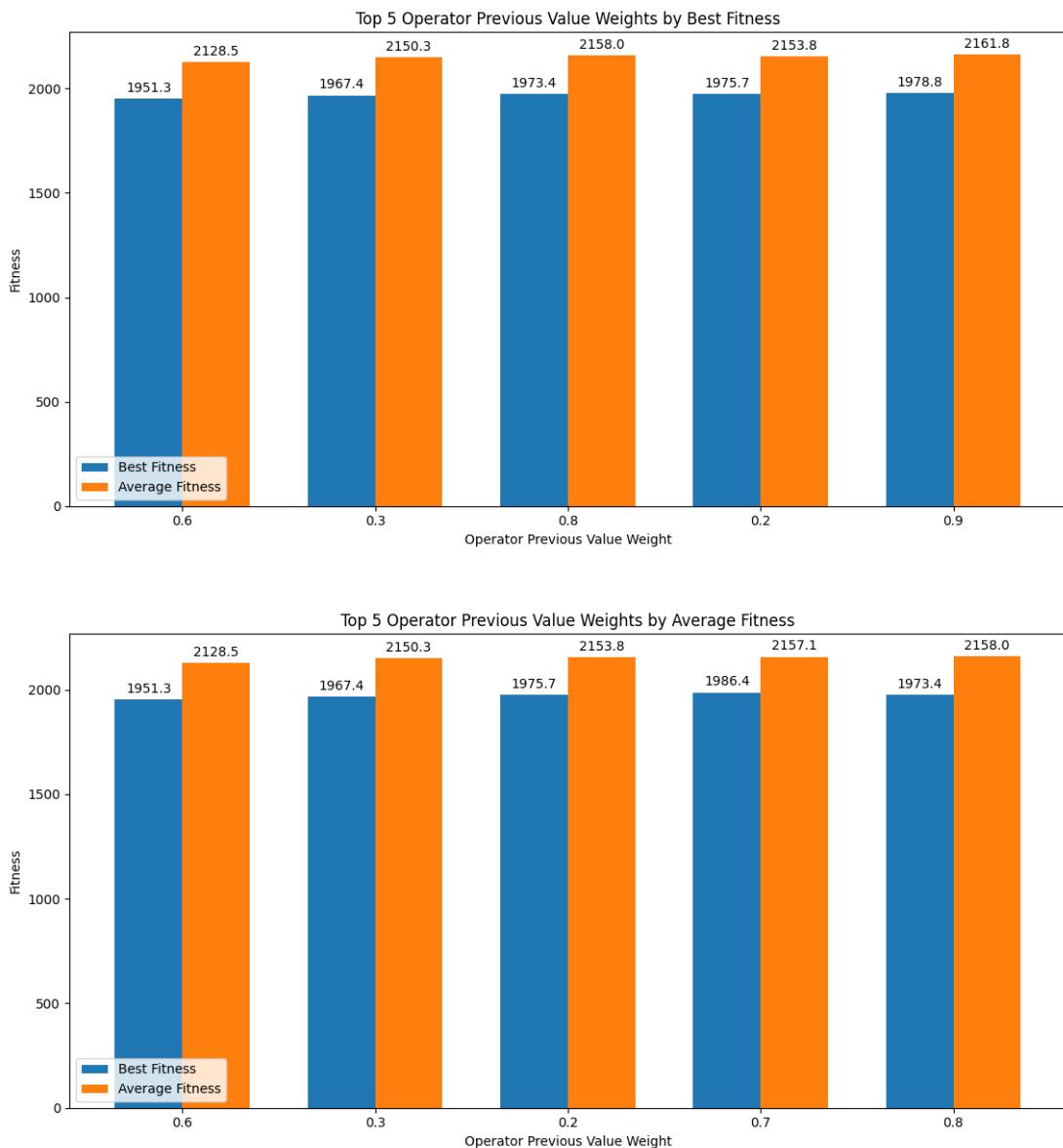
● ● ●

def update_weights(self):
    for i in range(len(self.operator_weights)):
        self.operator_weights[i] = (ALNS_PREV_WEIGHT *
                                    self.operator_weights[i]) + ((1-ALNS_PREV_WEIGHT) *
                                    (self.scores[i] / self.usages[i]))

```

בגלל שהבחנו בהשפעה הגדולה של החלוקת החזת על ביצועי האלגוריתם, הוחלט לעשות tuning שהשוואה בין חלוקות שונות מכפיליות של 0.1 וסכום 1 (למשל, אופציות אחרות היו הזוגות (0.7, 0.3) (0.1, 0.9) ו-





## סעיף 5

האלגוריתם ממומש ע"י המחלקה BranchAndBoundAlgorithm, המשמשת בעץ חיפוש לבניית הפתרון באמצעות הדרגתית. האלגוריתם זהה בשונה מהאלגוריתמים הקודמים ממומש בצורה שונה לכל אחת מהבעיות, אך תוך שימוש באותה לוגיקה של פירוק לבעיות קטנות ובניה הדרגתית של הפתרון.

**בבנייה ניתוב הרכבים**, הצומת בעץ מייצג פתרון חלקו המכיל את המסלולים שנבנו עד כה, כולל המסלול הנוכחי שבתהליך בנייתו, את הליקות הנכליים במסלולים הללו, את הליקות שנשארו, וחסם תחתון על העלות הסופית של הפתרון. מכאן שעלה הוא מצב בו כל הליקות נכללים במסלולים, והעלות המשוערת היא כעלוות האמיתית של הפתרון. כמו כן משתמשים

בעירימת מינימום העוזרת בלבוחר את המצב הכי מבטיח על מנת להתקדם ממנו, שהוא המצב בעל החסם התחתון הכי נמוך ברגע כלשהו.

שלבי האלגוריתם הם ככזה:

1. **מאתחלים עירימת מינימום עם המצב ההתחלתי** שהוא מצב בו אין מסלולים, כל הלקוחות עוד לא שובצו, והעלות של המסלול עד עכשו היא 0.

```
● ● ●  
  
initial_state = (0, [], customers, 0, [])  
queue = PriorityQueue()  
queue.put(initial_state)
```

2. כל עוד העירימה לא ריקה, **שולפים את המצב הכי מבטיח עירימת המינימום**:

```
● ● ●  
  
while not queue.empty() and  
    time.time() - elapsed_start_time < TIME_LIMIT:  
    _, route, remaining, cost, all_routes = queue.get()
```

- אם כל הלקוחות של המצב שובצו, אז יש פתרון מלא, ולכן מחשבים את העלות המלאה שלו, ואם הוא יותר טוב מהטוב ביותר שמצא עד כה הופכים אותו להיות הטוב ביותר.

```
● ● ●  
  
if not remaining:  
    total_routes = all_routes + [route]  
        if route else all_routes  
    total_cost = sum(self.estimate_cost(route)  
                    for route in total_routes)  
    if total_cost < self.best_cost:  
        self.best_cost = total_cost  
        self.best_solution = total_routes  
    continue
```

- אם יש ללקוחות שנשארו, מתנהגים לפי אם יש מסלול פתוח:

אם יש מסלול פתוח: מסתכלים על k (3) הלקוחות הקרובים ביותר ללקוח האחרון במסלול ולכל אחד בודקים אם ניתן להוסיף אותו אליו (אם קיibilitת הרכב מאפשרת), אם כן, מוסיפים אותו, אחרת סוגרים את המסלול הנוכחי.

אם אין מסלול פתוח: מוסיףים מסלול חדש עם הלקוות הבא בתור (שהוא הכי קרוב למבחן בין כל הלקוחות שנשארו).

```

● ● ●

if route:
    last_customer = route[-1]
    candidates = [c for c in self.k_nearest_neighbors[last_customer]
                  if c in remaining]
    if not candidates or len(candidates) == 1:
        candidates = self.compute_k_nearest_neighbors(k=2,
                                                       customer=last_customer, other_customers=remaining)[last_customer]
    else:
        candidates = remaining

```

### כ. מחשבים את החסם התחתיו החדש של הפתרון, ואם הוא טוב מספיק

(קטן מהפתרון המלא הטוב ביותר שנמצא עד עכשוו) **שומרים אותו בערימה.**

чисוב החסם התחתיו מתבצע לפי אם יש מסלול פתוח או שלא. אם יש אז הוא יהיה סכום של המרחקים של המסלולים כולל הנוכחי, בנוסף לעצ הפרש המינימלי של הקואורדינטות של הערים שנשארו. אם אין מסלול פתוח אז הוא מחושב באותה שיטה, רק שבמקרה המסלול הנוכחי שלכורה נסגר מחושב מסלול דמה שמכיל את העיר שהייתה אמורה להיכנס למסלול, דבר זה נעשה בשל כך שבוואדי יהיה מסלול נוסף שיתחיל ויחזור למיחס הכלול את הציגת ההייא.

```

● ● ●

for customer in candidates:
    i = remaining.index(customer)
    new_route = route + [customer]
    new_demand = sum(self.population.demands[c] for c in new_route)
    new_remaining = remaining[:i] + remaining[i+1:]

    if new_demand <= self.population.truck_capacity:
        partial_routes = all_routes.copy()
        partial_cost = cost
        lower_bound = partial_cost + self.estimate_cost(new_route)
                     + self.mst_estimate(new_remaining)
        if lower_bound < self.best_cost:
            queue.put((lower_bound, new_route, new_remaining,
                       partial_cost, partial_routes))
    else:
        if route:
            partial_routes = all_routes + [route]
            partial_cost = cost + self.estimate_cost(route)
            lower_bound = partial_cost + self.estimate_cost([customer])
                         + self.mst_estimate(new_remaining)
            if lower_bound < self.best_cost:
                queue.put((lower_bound, [customer], new_remaining,
                           partial_cost, partial_routes))

```

איור 14: קטע הקוד האחראי על חישוב החסם התחתיו לתת הפתרון.

```

def estimate_cost(self, route):
    if not route:
        return 0
    cost = np.linalg.norm(np.array(self.population.coords[route[0]]))
    - np.array(self.population.depot))
    cost += sum(self.population.dist_matrix[route[i]][route[i + 1]]
                for i in range(len(route) - 1))
    cost += np.linalg.norm(np.array(self.population.coords[route[-1]]))
    - np.array(self.population.depot))
    return cost

```

איור 15: הפקציה האחראית על חישוב עלות הפתרון שנבנה עד כה.

```

def mst_estimate(self, remaining_customers):
    if not remaining_customers:
        return 0
    coords = [self.population.coords[cid]
              for cid in remaining_customers]
    n = len(coords)
    visited = [False] * n
    min_edge = [float('inf')] * n
    min_edge[0] = 0
    total = 0
    for _ in range(n):
        u = -1
        for i in range(n):
            if not visited[i] and (u == -1
                                   or min_edge[i] < min_edge[u]):
                u = i
        visited[u] = True
        total += min_edge[u]
        for v in range(n):
            if not visited[v]:
                dist = np.linalg.norm(np.array(coords[u])
                                      - np.array(coords[v]))
                if dist < min_edge[v]:
                    min_edge[v] = dist
    return total

```

איור 16: הפקציה האחראית על בניית העץ הפורש המינימום של הערים שעוד לא נכללו בפתרון.

בשלב ס בחלק "אם יש מסלול פתוח" היינו לוחכים בחשבון את כל הלקוחות שנשארו ומנסים לשרשר אותם ללקוח האחרון במסלול, אולם זה עלה בסיבוכיות זמן גבוהה מאד, מכאן בא הרעיון של להסתפק ב- k השכנים הקרובים ביותר, דבר שמחית סיבוכיות ומקס את האלגוריתם לכיוון פתרונות בעלי פוטנציאל גדול. הערך של k נקבע אותו להיות 3 כברירת מחדל, שיאזן בין סיבוכיות הזמן לבין יכולת הפתרון המתאפשר, אולם בהיותו משתנה ניתן לשנות את הערך שלו בהתאם לצורך.

```

● ● ●

def compute_k_nearest_neighbors(self, k=2, customer=None, other_customers = None):
    neighbors = {}
    all_customers = list(self.population.coords.keys())
        if customer is None else [customer]
    all_other_customers = list(self.population.coords.keys())
        if other_customers is None else other_customers
    for i in all_customers:
        distances = []
        for j in all_other_customers:
            if i != j:
                dist = self.population.dist_matrix[i][j]
                distances.append((dist, j))
        distances.sort()
        neighbors[i] = [j for _, j in distances[:k]]
    return neighbors

```

איור 17: הפקציה שモוצאת לכל עיר את  $k$  הערים הקרובים אליה מתוך אוסף הערים הנתונים, או מתוך כל הערים במרקחה שלא נתנו אוסף ערים כארוגמן.

**בבעיית Ackley**, המצב כולל רשימה של ערכים של הממדים שכבר נבחרו, את מספר הממדים שכבר נבחרו, חסם תחתון על העלות של הווקטור (הפתרון) הסופי, וכן עליה הוא מצב בו מספר הממדים שנבחרו הוא מספר הממדים בפתרון הסופי (שהוא 10 במקרה שלנו). האלגוריתם הוא כמוポート להלן:

1. **מאותלים ערימת מינימום עם המצב ההתחלתי** שהוא מצב דמה בו החסם תחתון הוא אפס, וווקטור ריק, ומספר הממדים שנבחרו הוא אפס.

```

● ● ●

initial_state = (0, [], 0)
queue = PriorityQueue()
queue.put(initial_state)

```

2. כל עוד הערימה לא ריקה, **שולפים את המצב הכי מבטיח**:

```

● ● ●

while not queue.empty() and time.time()
    - elapsed_start_time < TIME_LIMIT:
        bound, vector_so_far, var_idx = queue.get()

```

לאחר מכן, מתנהגים לפי אם מספר הממדים שנבחרו הוא מספר הממדים הרצוי בפתרון הסופי (10 במקרה שלנו):

אם כן, מחשבים את העלות האמיתית של הווקטור, ואם הוא יותר טוב מהטוב ביותר, הופכים אותו להיות הטוב ביותר.

```
if var_idx == dim:  
    vector = np.array(vector_so_far)  
    fitness = dummy_individual.evaluate(vector)  
    if fitness < self.best_cost:  
        self.best_cost = fitness  
        self.best_solution = vector.copy()  
    continue
```

אם לא, מפרקם את הטווח [-32.768, 32.768] ל- 100 תת-טווחים, לוקחים באופן רנדומלי ערך מכל תת-טווח, מוסיפים אותו לוקטור ומחשבים את החסם הנוכחי על הפתרון הסופי, ואם הוא טוב מספק (הכי טוב שנמצא עד כה) מוסיפים אותו לעירימה.

```
if var_idx == dim:  
    vector = np.array(vector_so_far)  
    fitness = dummy_individual.evaluate(vector)  
    if fitness < self.best_cost:  
        self.best_cost = fitness  
        self.best_solution = vector.copy()  
    continue  
  
subdivisions = 100  
step = (upper_bound - lower_bound) / subdivisions  
  
for i in range(subdivisions + 1):  
  
    xi = random.uniform(lower_bound  
                        + i * step, lower_bound + (i + 1) * step)  
    new_vector = vector_so_far + [xi]  
  
    partial_vector = np.array(new_vector  
                            + [0] * (dim - len(new_vector)))  
    partial_cost = dummy_individual.evaluate(partial_vector)  
  
    if partial_cost < self.best_cost:  
        queue.put((partial_cost, new_vector, var_idx + 1))
```

чисוב החסם הנוכחי על העלות מתבצע ע"י ריפוד אפסים, קלומר בהנחה שכל שאר הממדים בעלי ערך 0.

היררכיות שנעשתה בהם שימוש במעבדה

- הዮристיקת בניית הפתרון ההתחלתי:** נעשה בה שימוש באמצעות הפונקציה Multi-Stage cvrp\_generate\_assignment המשמשת אלגוריתמים רבים, ביניהם Heuristics Algorithm לאותחול אוכלוסיה ההתחלתית. ההיוurstיקת מבוססת על קר שערים קרובות אחת לשניה כדי לבקר ביחיד, וזה לא יוסיף הרבה אורך, וכן הפענקציה, ששואבת את הרעיון שלה מאלגוריתם k-means, מנסה לחלק את הערים לקבוצות שמספרם עד מספר הרכבים על בסיס מיקום גאוגרפי קר שהערים באותו קבוצת יכללו באותו מסלול.

**סיבוכיות:** בכל איטרציה יש לנו  $N$  לקובוחות, ומחשבים את המרחק שלהם מ-  $K$  מרכזיים, וכל חישוב לוקח  $O(1)$  זמן. יש לנו 4 איטרציות בסה"כ שזה קבוע. וכן הסיבוכיות הכלולת היא  $(K * N)O$ .
- הዮurstיקת סידור הלקוחות בתוך המסלול:** נעשה בה שימוש באלגוריתם Multi-Stage Heuristics, בשלב שאחרי אתחול האוכלוסיה, במטרה לקבוע את סדר הערים בתוך המסלול. ההיוurstיקת מtabסת על קר שבכל צעד העיר הסבי ביוטר שתהיה הבאה בתוך היא הכי קרובה.

**סיבוכיות:**  $(N^2)O$  כאשר  $N$  הוא מספר הערים בבעיה. בכל פעם אנחנו עוברים על הערים, מחשבים את המרחק שלה מהעיר הנוכחית מהמחсон ולוקחים את הקרובה ביותר. (ספציפית באלגוריתם שלנו, אנחנו מחשבים את כל המרחקים בין כל עיר ועיר בהתחלה, בונים מטריצה וממש שולפים ממנה כל פעם את המרחקים שנרצה).

המצב הגרוע ביוטר הוא כאשר כל הערים נמצאות על אותו מסלול.
- העץ הפורש מינימלי (MST) בין הקואורדינטות של הערים כחסם תחתון על אורך המסלולים:** באלגוריתם Branch and Bound, במהלך בניית הפתרון, שמתבצעת באופן הדרגי, אנחנו מחשבים חסם תחתון על אורך המסלולים שעוז לא נבנו ע"י חישוב העץ הפורש מינימלי בין הערים שעוז לא נכללו בתוך המסלולים הקיימים עד כה. הדבר מאפשר לבדוק אם כדי להמשיך בחיפוש בתת העץ של המצב או לעשות גיזום ולהסוך במשאים.

**סיבוכיות:** משתמשים באלגוריתם Prim, קר שבכל איטרציה אנחנו בוחנים את הערים שלא ביקרנו בהם ולוקחים את המינימלי מביניהם שזה  $(N)O$  במצב הגרוע, כאשר  $N$  הוא מספר הערים הבודקים. מעדכנים את הקשתות שמקשרות הבודקים אותן שכנו שזה גם  $(N)O$ . ומן שיש לנו מספר איטרציות כמספר הבודקים אז הסיבוכיות הכלולת היא  $(N^2)O = (N)O * (N + N)O$ .
- בדיקה 3 הערים הקרובות ביותר כפונצייאלים להיות העיר הבאה במסלול:** משתמשים בו באלגוריתם Branch and Bound, במהלך בניית הפתרון שנעשה באופן הדרגי ובאמצעות עז, אנחנו מנסים לפתח את הצומת (המייצג מצב) שזה

שקלל לעיר הבאה בתור אחרי העיר האחרונה במסלול הפתוח, ע"י כך שאנו חנו בודקים את האופציית של 3 הערים הקרובות ביותר, על סמך כך שערים קרובות כדאי לבדוק ביחד. כמו כן, חלק מההיררכיה, נעשה סידור לערים לפי מרחקם מהמחсон בהתחלה האלגוריתם במטרה שכשנרצה לפתח מסלול העיר שניקח תהיה הקרובה ביותר למיחסן מבין אלה שעוז לא שובצו.

סיבוכיות: שליפת המרחקים של העיר האחרונה במסלול ב-  $O(N)$  כמספר הערים,

מיונם מתבצע ב-  $(N * \log N)$ , הדבר נעשה ל-  $N$  לקוות, لكن בסה"כ:

$$O(N^2 * \log N)$$

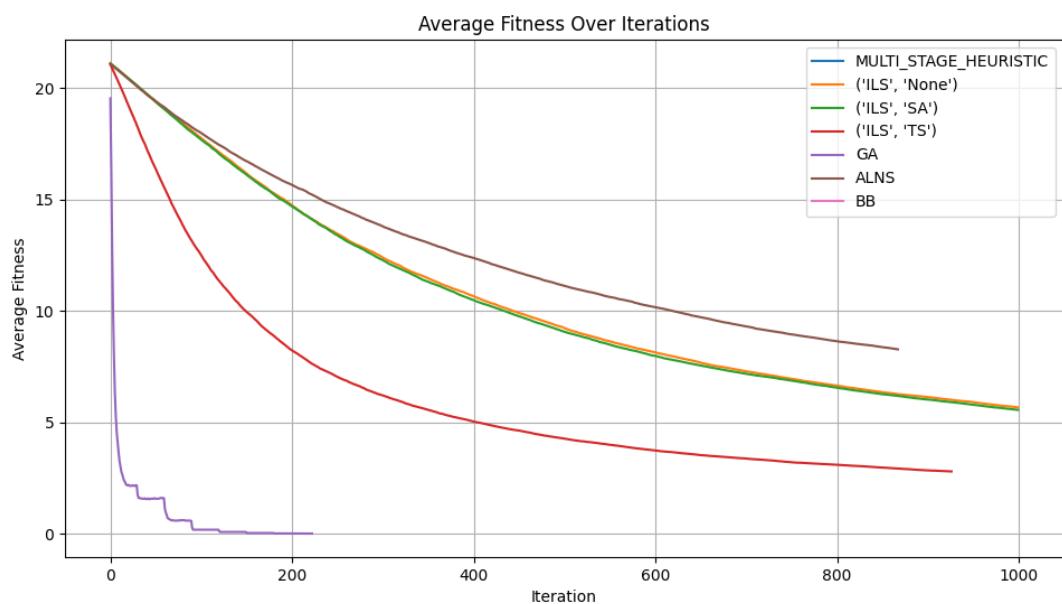
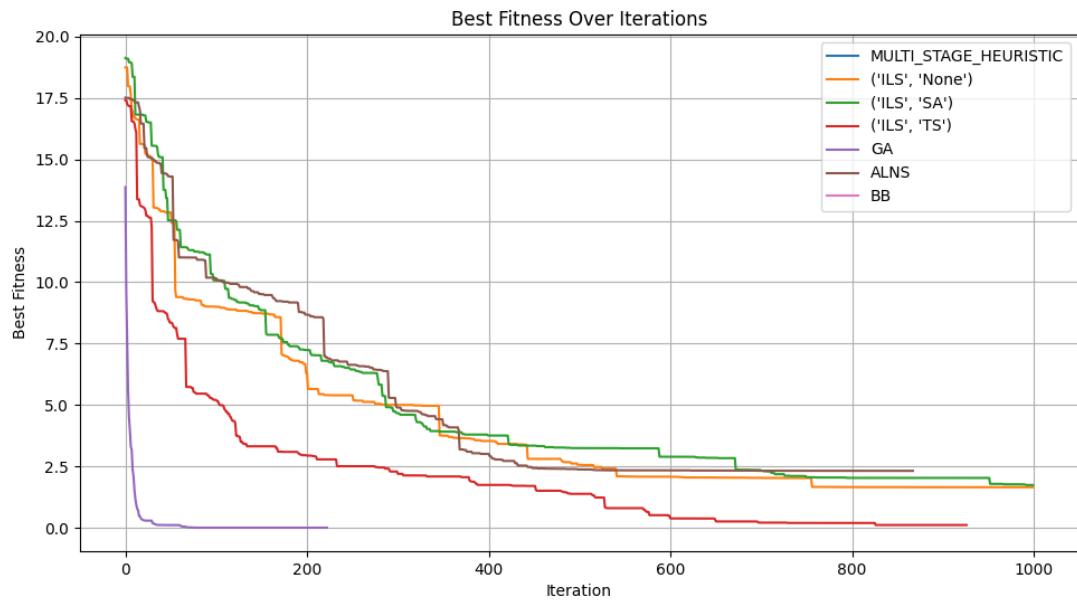
### ההשוואה בין ביצועי האלגוריתמים על הקלטים

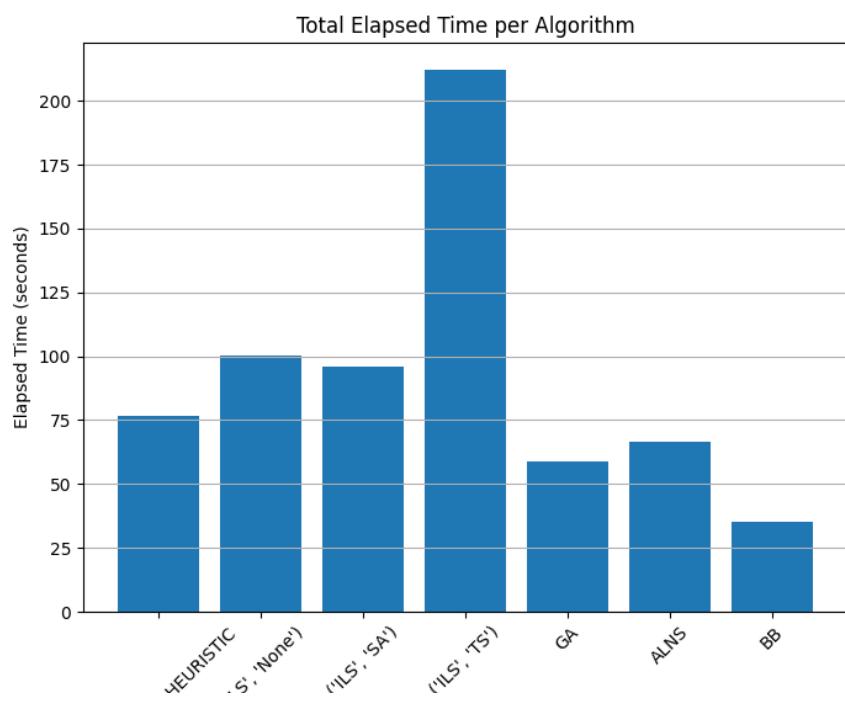
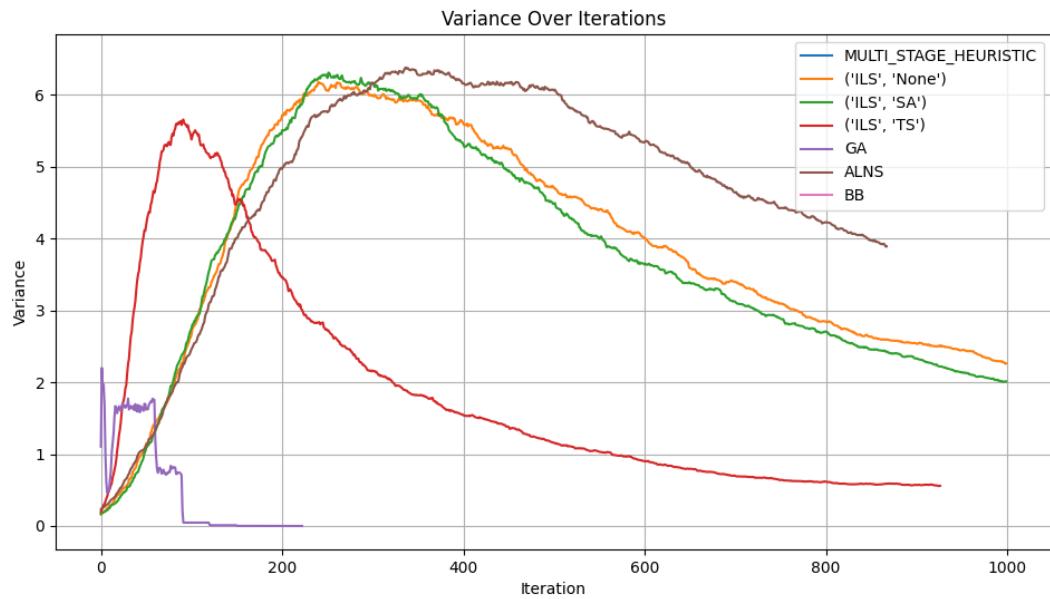
להלן תוצאות ההשוואה בין ביצועי האלגוריתמים על כל אחד מהקבצים הנתונים כקלט, בנוסף לקלט של בעיית Ackley. בחרנו להשוות בין האלגוריתמים לפי הפיטניס (העלות) הći גבוהה שהפיך, ממוצע הפיטניסים, השונות בין הפיטניסים, וזמן הביצוע.

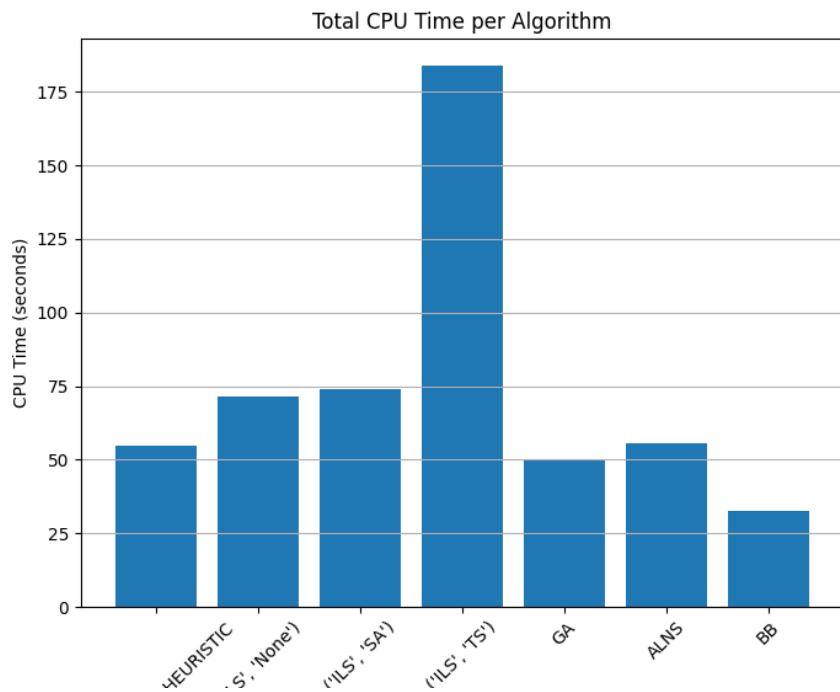
הערות:

- בשל המבנה הלא איטרטיבי של שני האלגוריתמים Branch and Multi-Heuristic ו- Branch and Bound הם לא מופיעים ב- 3 הגרפים הראשונים המראים את הביצוע לאורח האיטרציות.
- בטבלה האחורונה, המספרים של הפיטניס הטוב ביותר, הממוצע, והשונות מתychisms לערכים בתום ריצת האלגוריתם.
- בשל לחץ הזמן, עברו חלק מהקלטים (חלק מה- 4 בסדר כאן), הוריצה גרסה מצומצמת של אלגוריתם Branch and Bound, בה עבר על עיר היא מחפשת את השcn הכי קרוב בלבד, במקום 3, שכן התוצאות לא בהכרח משקפות את ביצועי האלגוריתם שבפועל יותר טובים ולא את זמן הריצה שבפועל יותר ארוך.

**:Ackley בעיית**

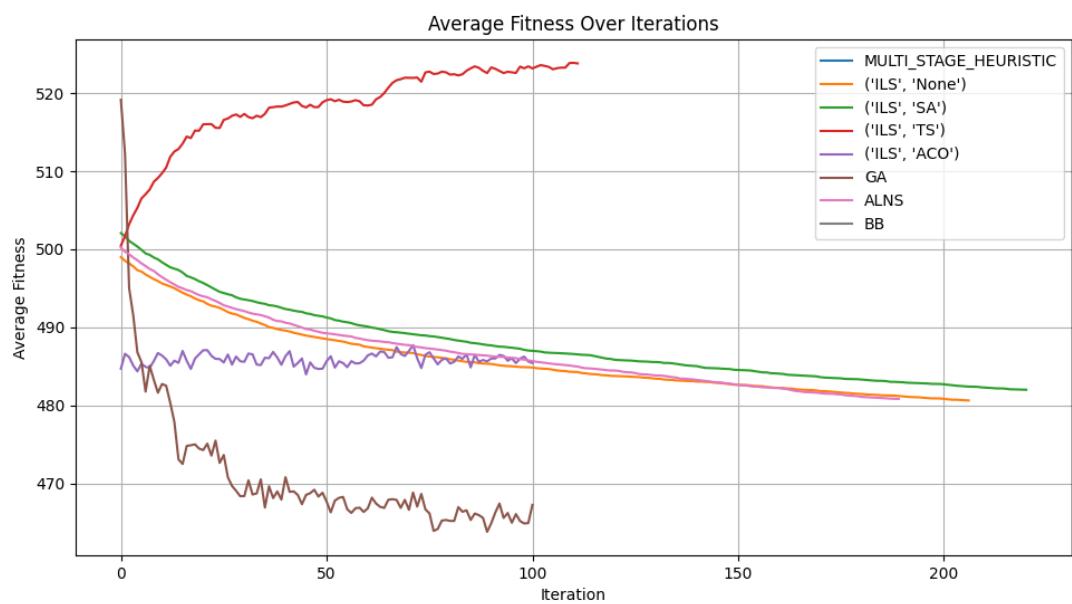
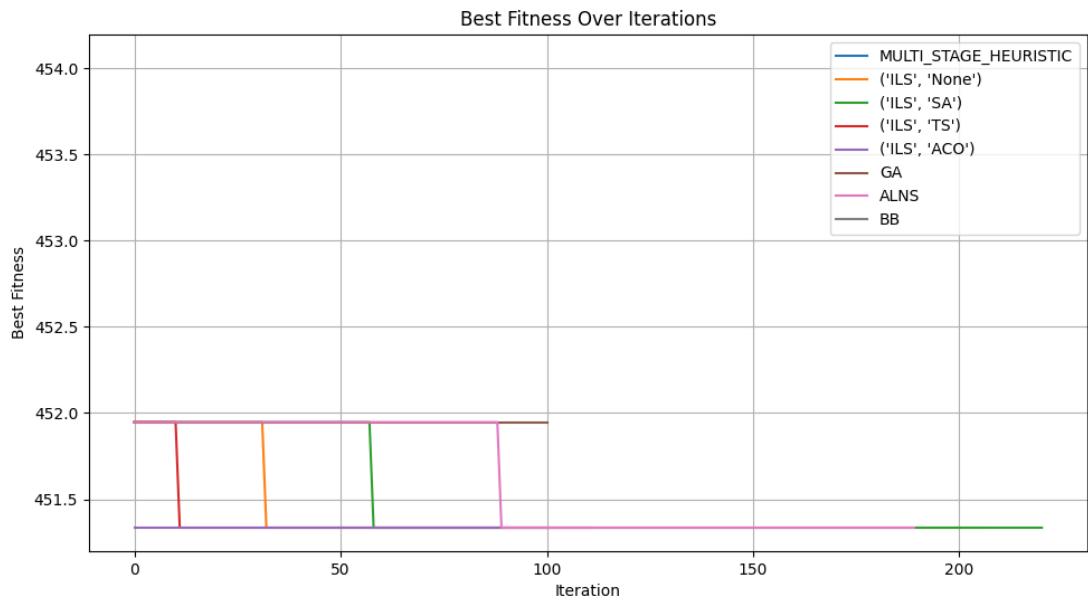


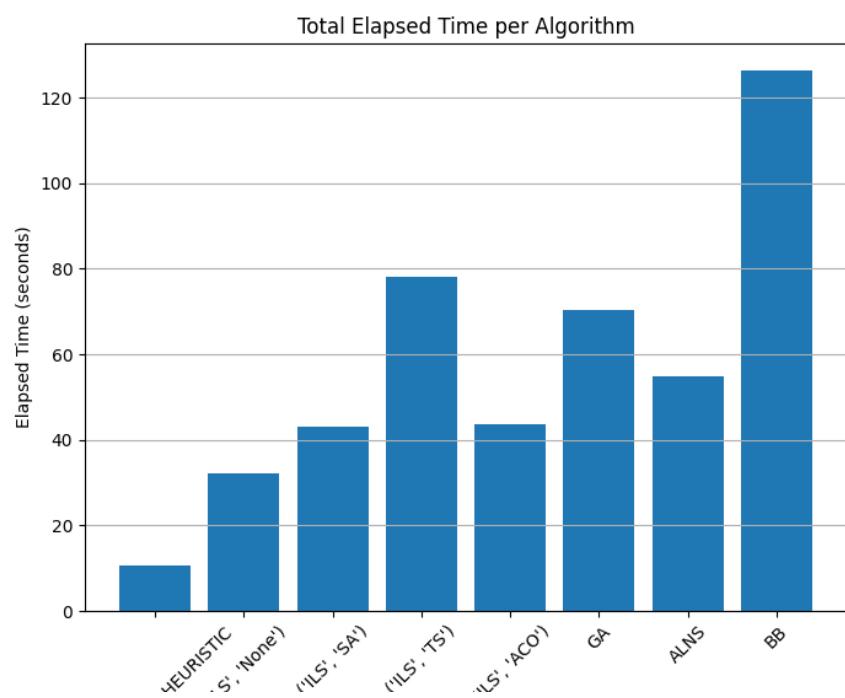
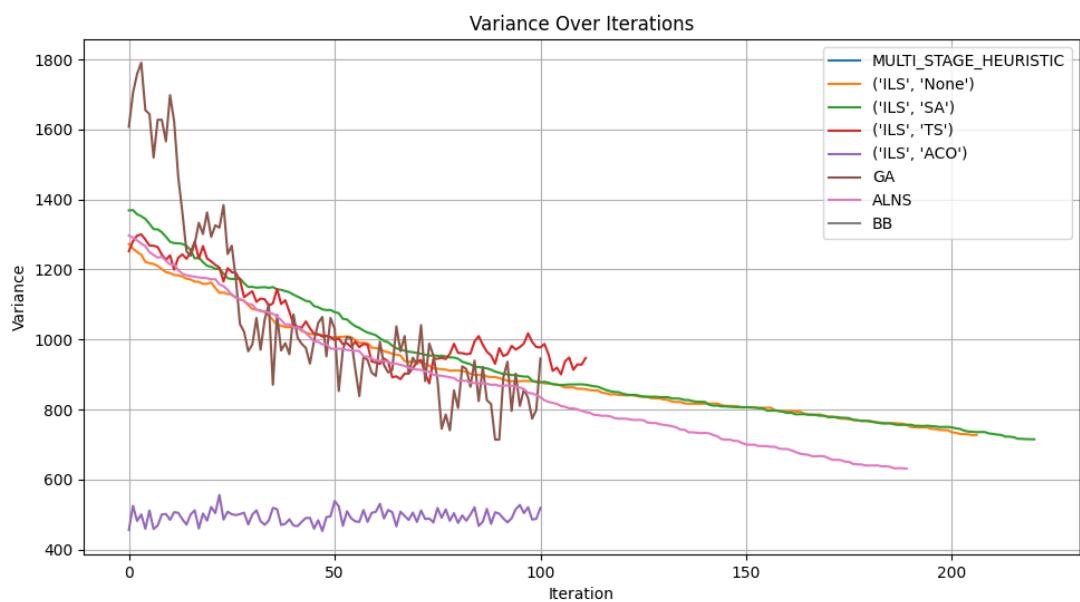


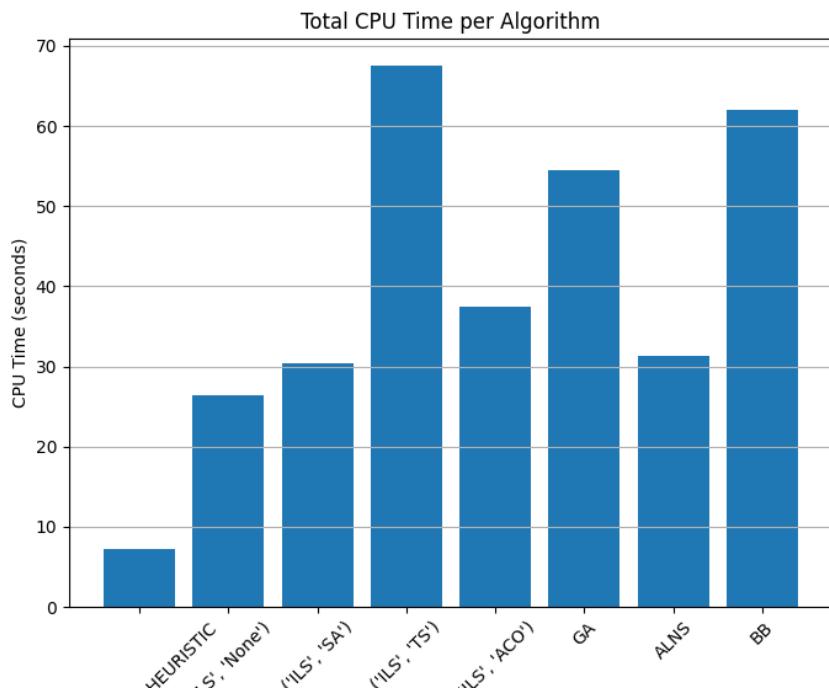


Algorithm	Best Fitness	Average Fitness	Variance	Elapsed Time	CPU Time
MULTI_STAGE_HEURISTIC	2.88	19.73	6.52	76.44	54.53
('ILS', 'None')	1.65	5.67	2.26	100.26	71.51
('ILS', 'SA')	1.74	5.55	2.01	96.04	74.13
('ILS', 'TS')	0.11	2.79	0.56	212.14	183.85
GA	0.00	0.00	0.00	58.84	50.15
ALNS	2.32	8.27	3.89	66.49	55.58
BB	2.18	-	-	35.00	32.72

:P-n16-k8

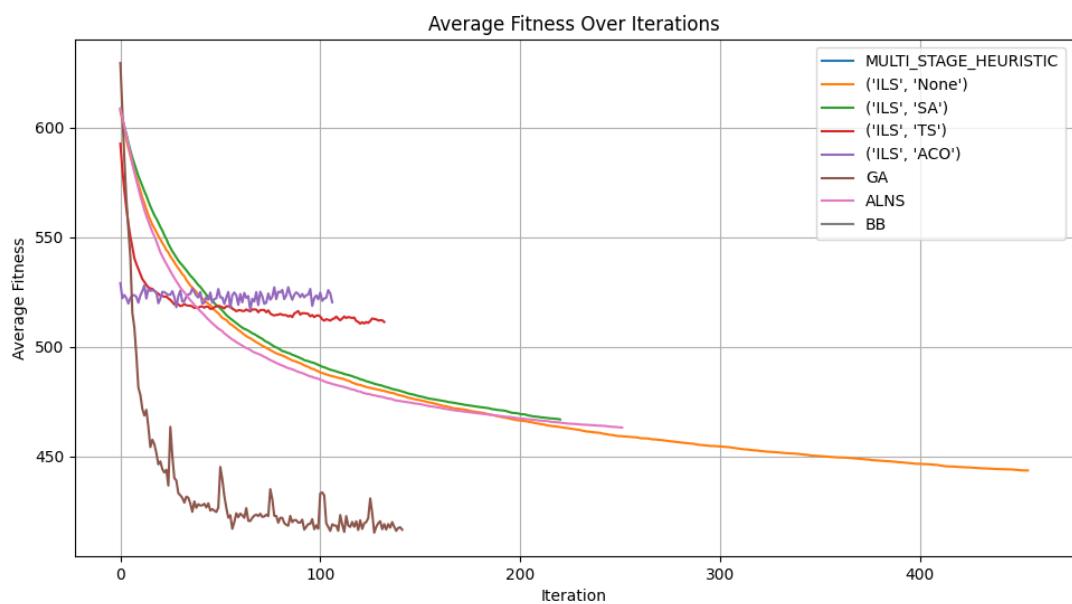
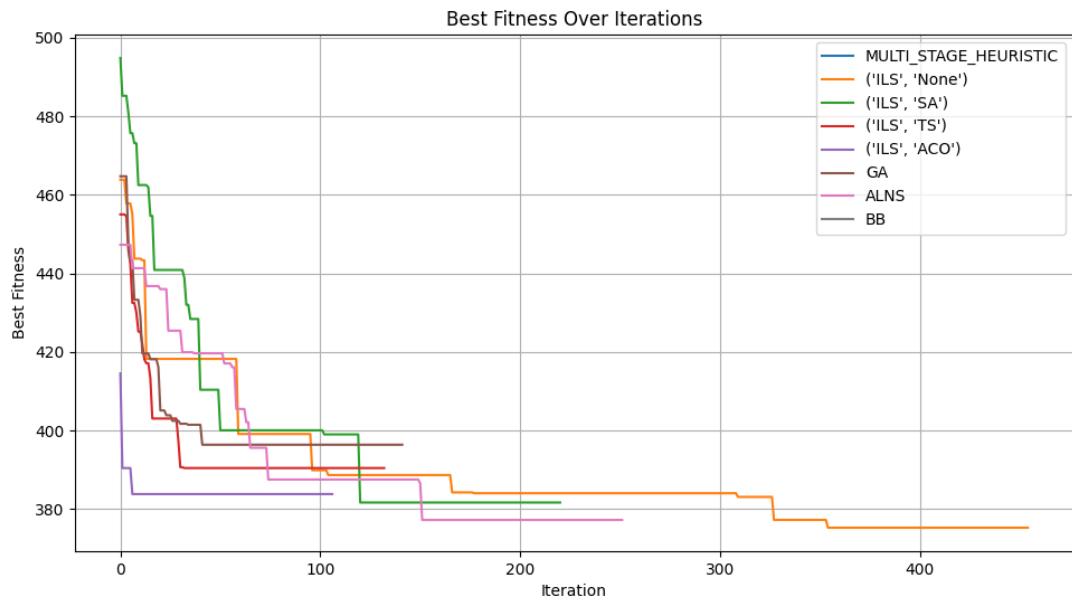


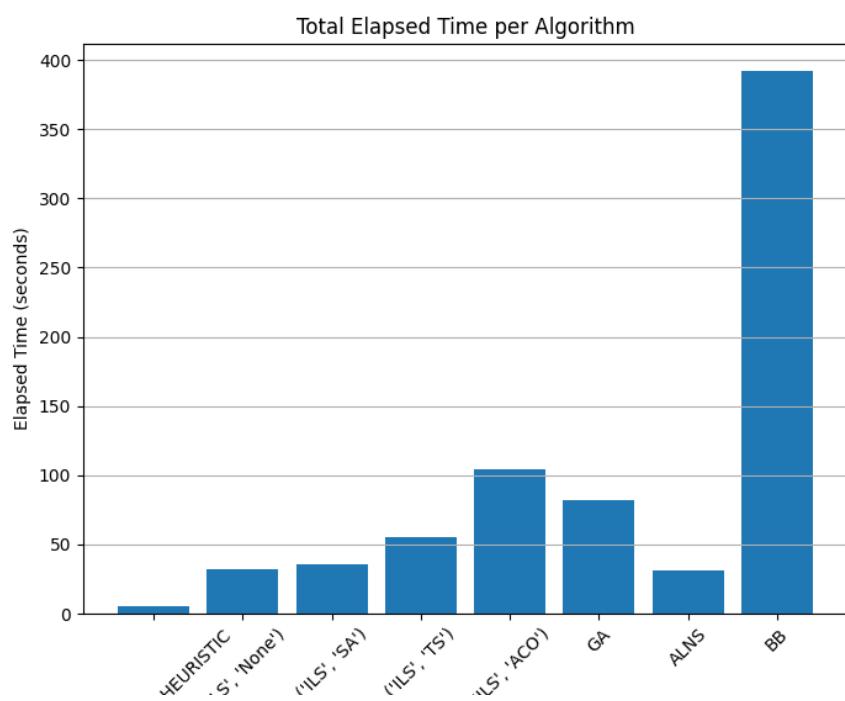
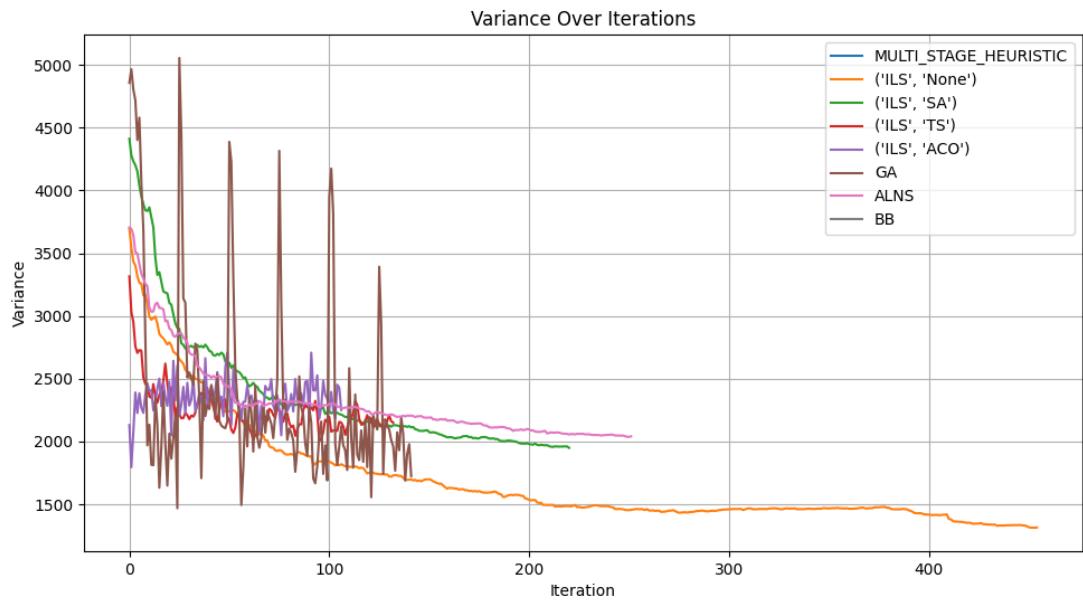


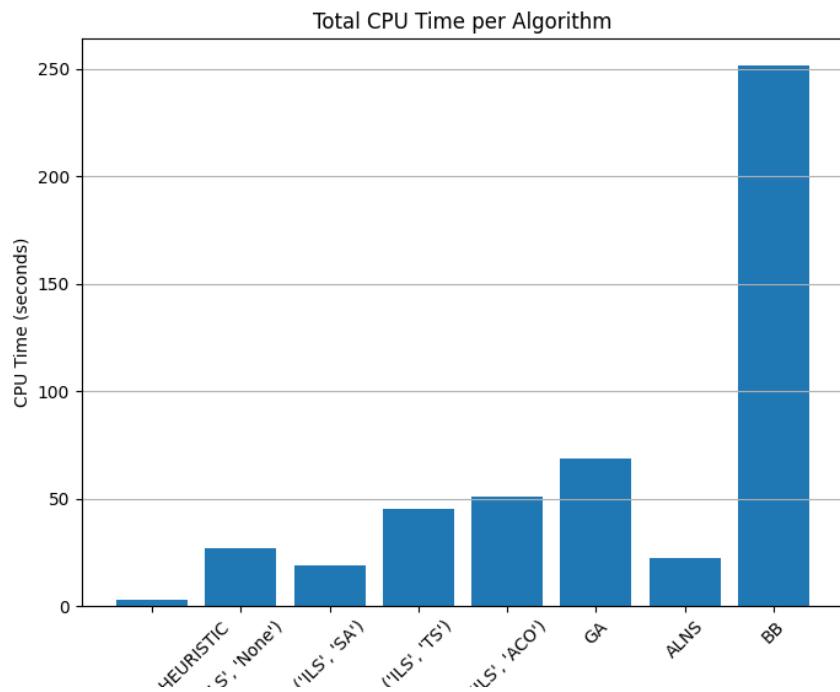


Algorithm	Best Fitness	Average Fitness	Variance	Elapsed Time	CPU Time
MULTI_STAGE_HEURISTIC	454.06	496.13	1176.85	10.60	7.18
('ILS', 'None')	451.34	480.62	727.55	32.08	26.33
('ILS', 'SA')	451.34	482.00	715.11	43.09	30.41
('ILS', 'TS')	451.34	523.81	947.27	78.07	67.54
('ILS', 'ACO')	451.34	485.42	519.17	43.70	37.41
GA	451.95	467.25	945.75	70.38	54.52
ALNS	451.34	480.82	630.99	54.79	31.28
BB	451.34	-	-	126.38	62.00

:E-n22-k4

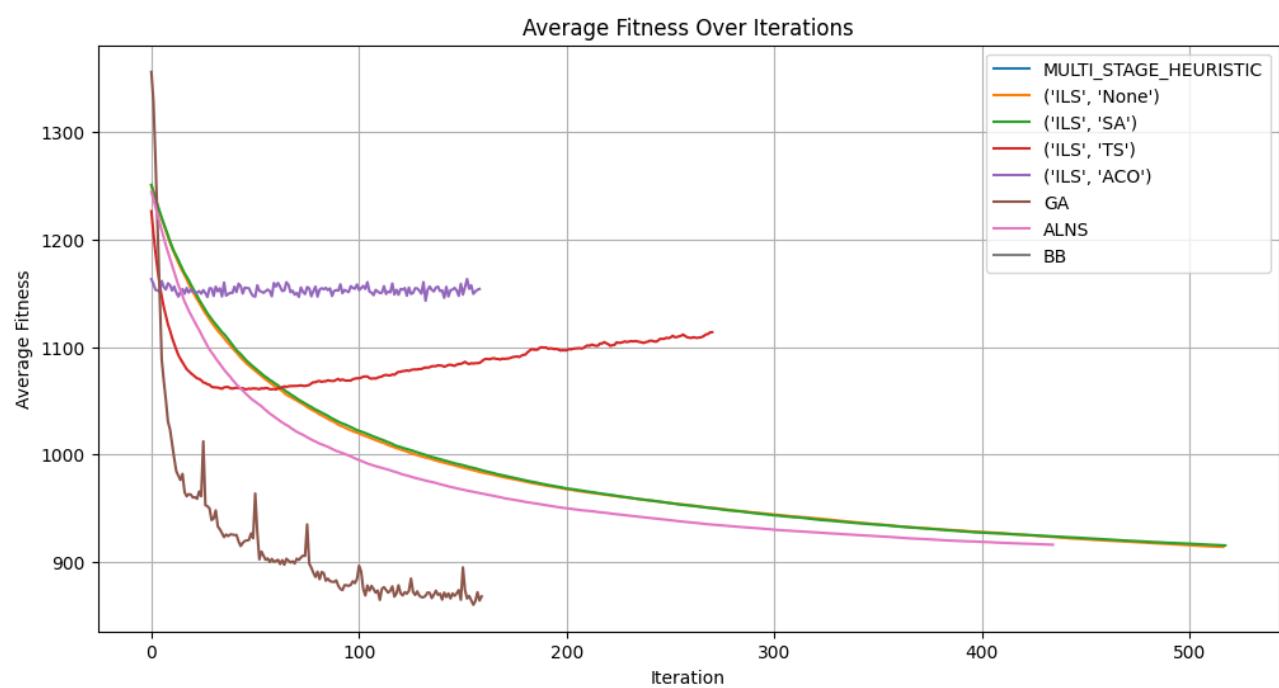
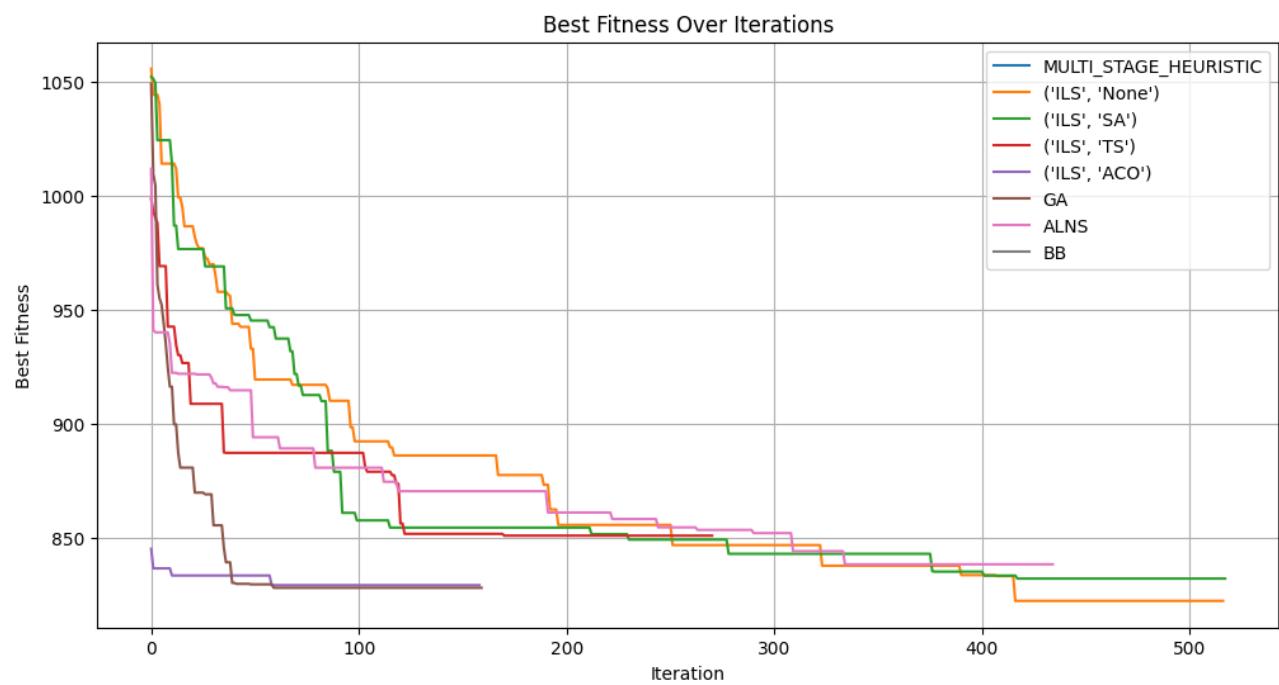




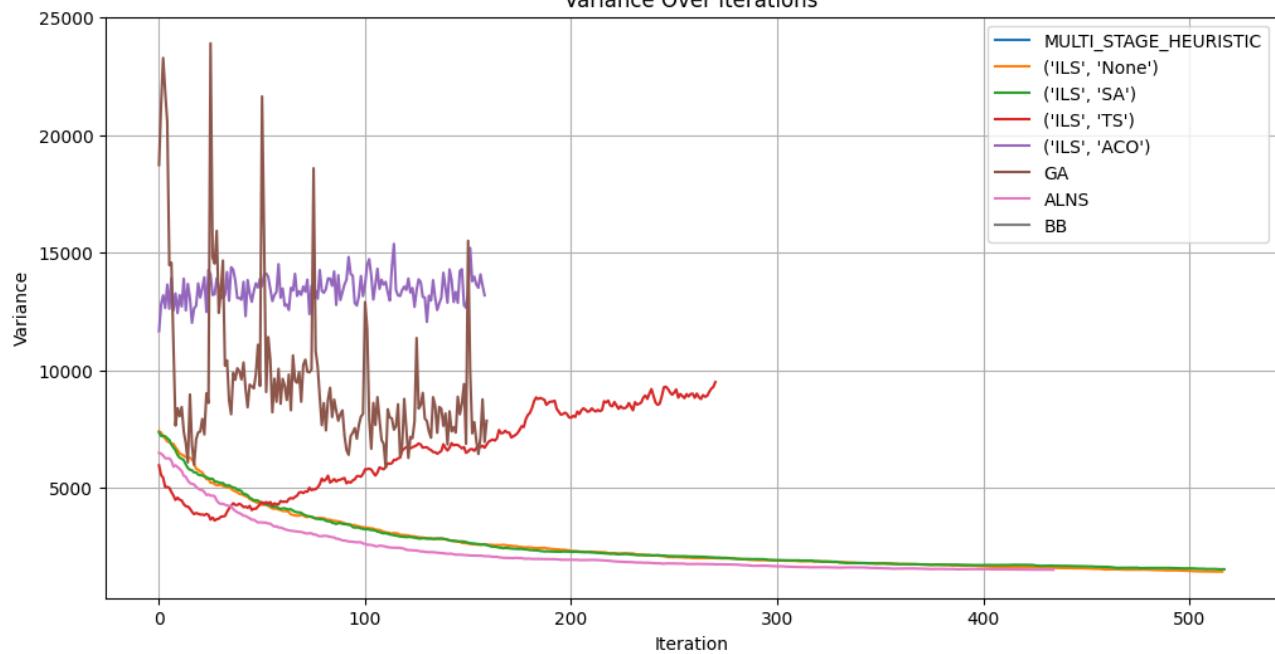


Algorithm	Best Fitness	Average Fitness	Variance	Elapsed Time	CPU Time
MULTI_STAGE_HEURISTIC	402.91	489.87	2375.57	4.95	2.65
('ILS', 'None')	375.28	443.62	1315.79	32.03	27.06
('ILS', 'SA')	381.69	466.88	1949.70	35.46	19.01
('ILS', 'TS')	390.47	511.37	2150.96	55.47	45.10
('ILS', 'ACO')	383.84	520.32	2251.34	104.08	50.86
GA	396.40	416.61	1724.60	81.89	68.77
ALNS	377.26	463.20	2041.34	31.37	22.15
BB	379.41	-	-	392.02	251.68

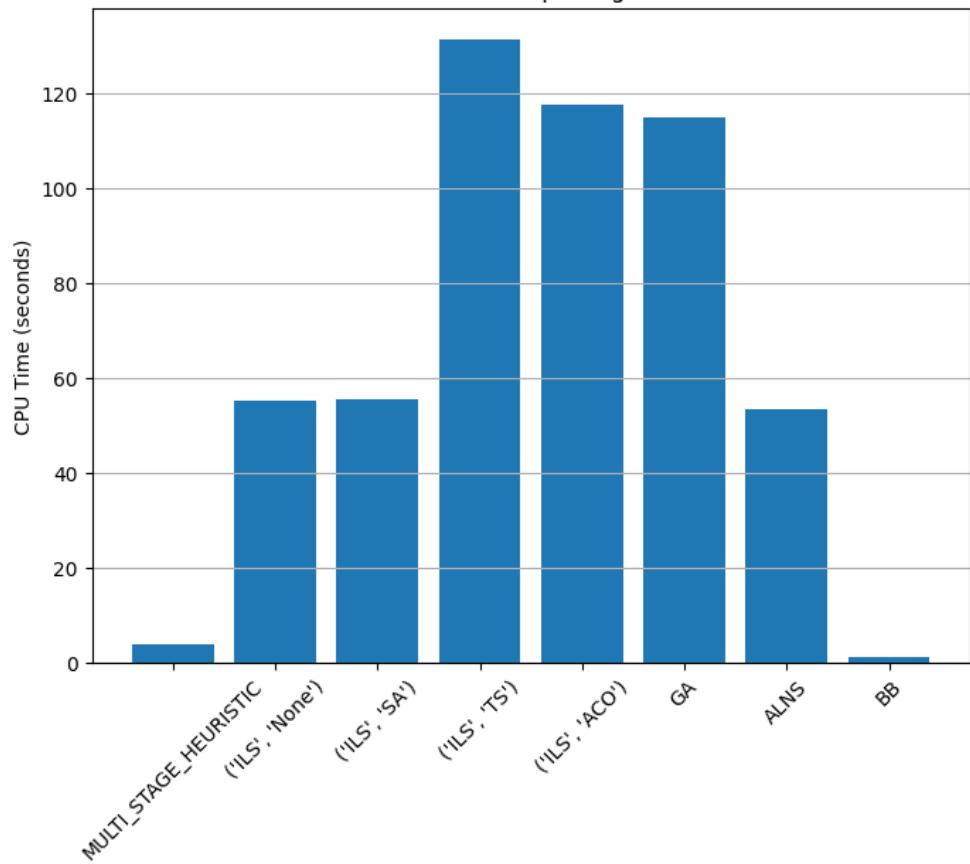
:A-n32-k5



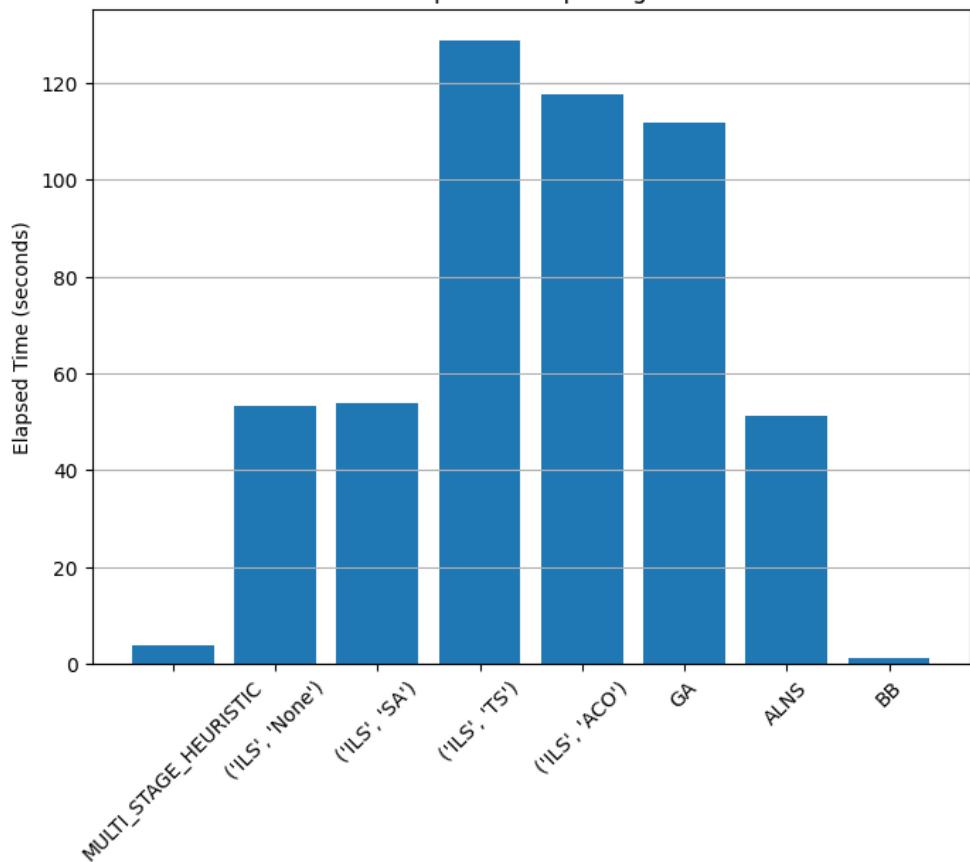
Variance Over Iterations



Total CPU Time per Algorithm

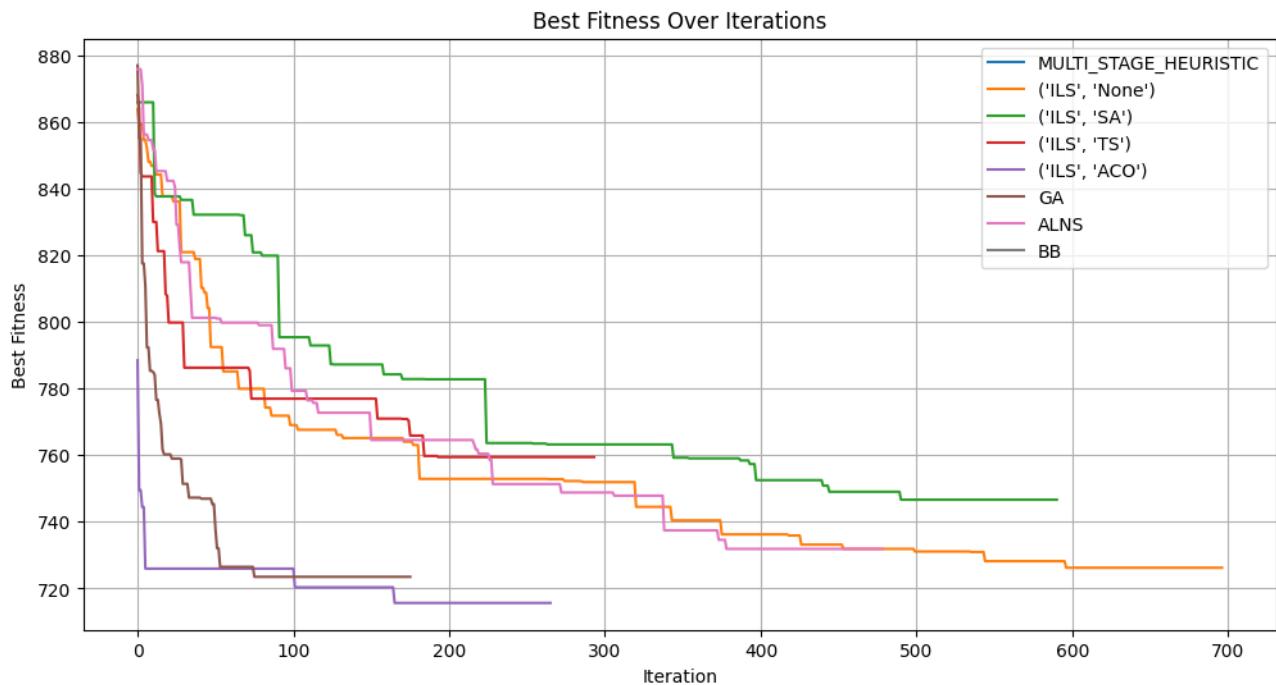


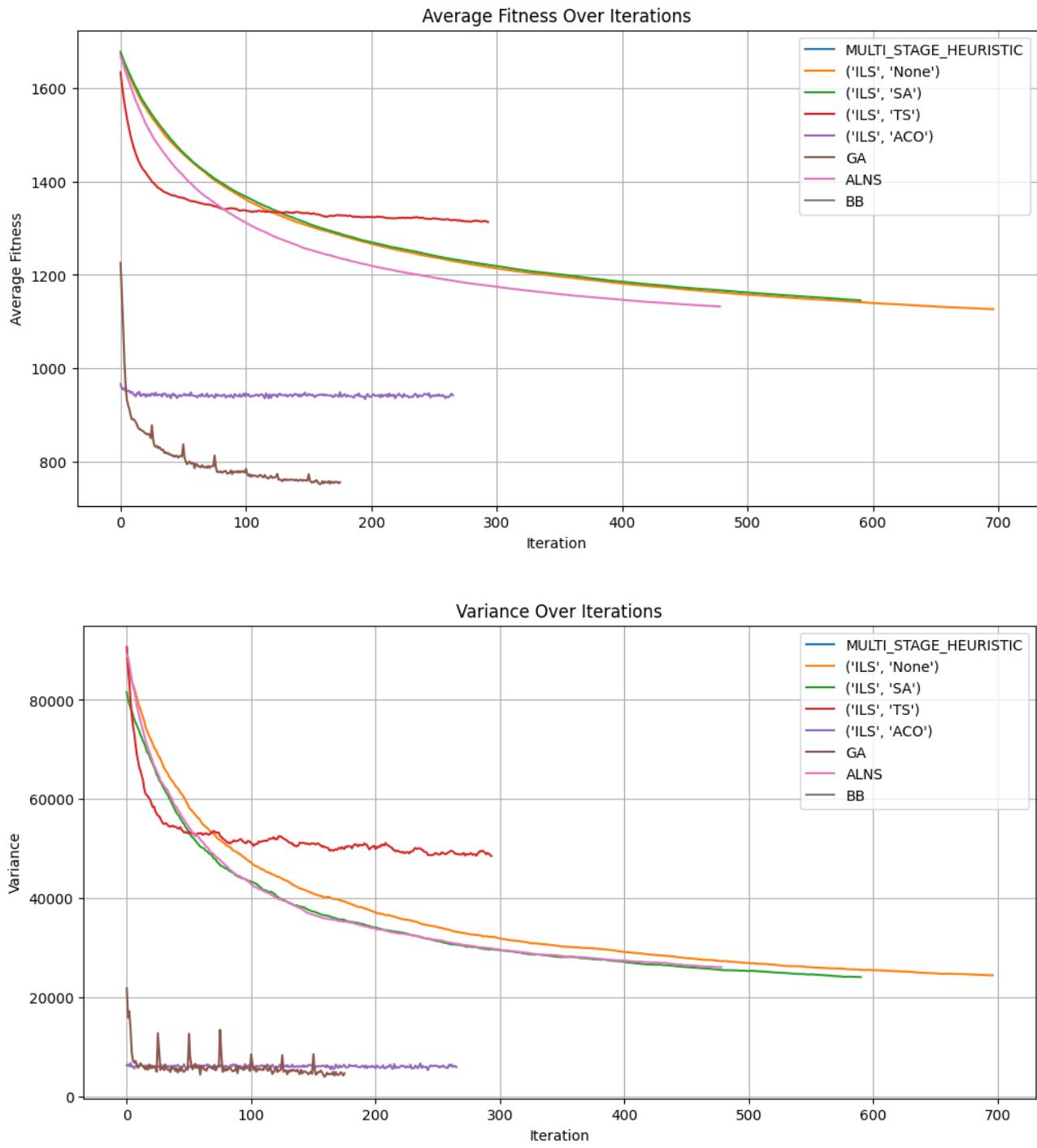
Total Elapsed Time per Algorithm

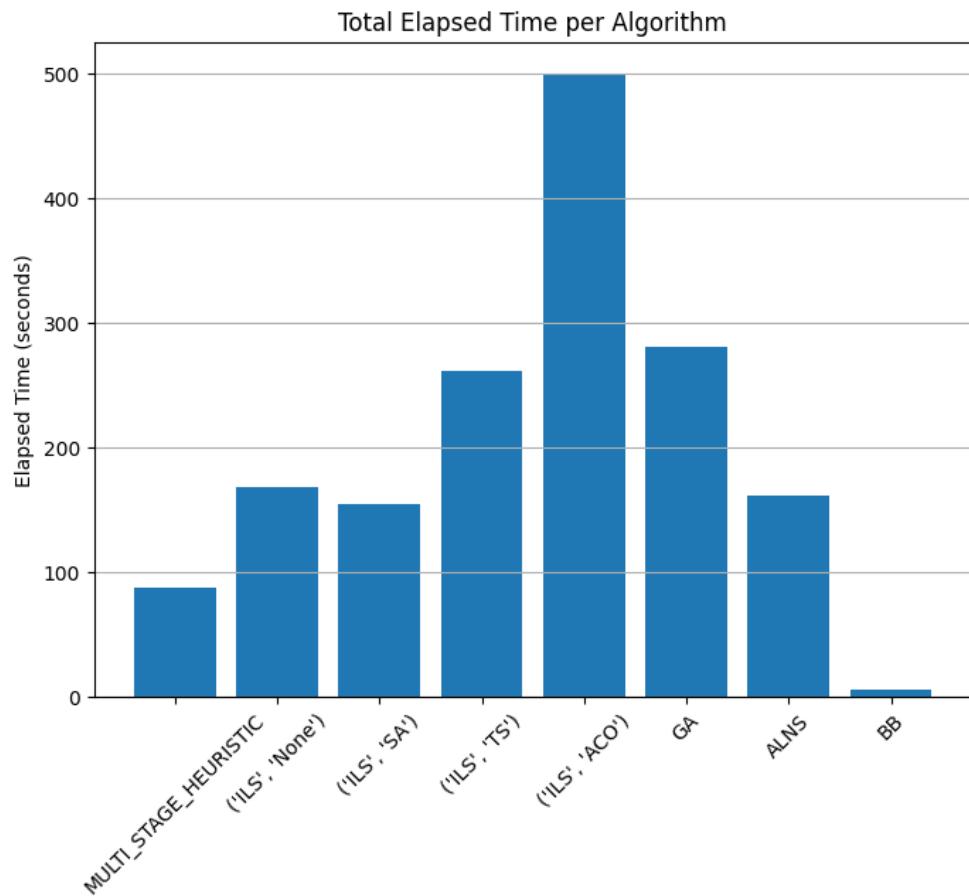


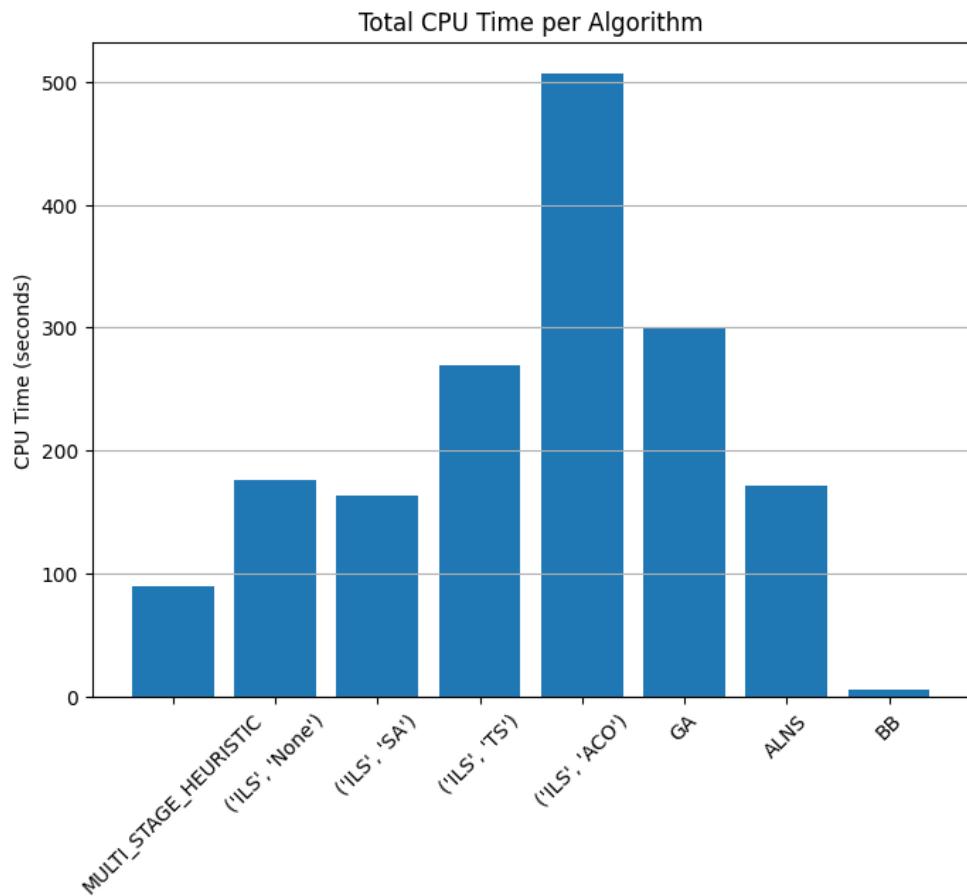
Algorithm	Best Fitness	Average Fitness	Variance	Elapsed Time	CPU Time
<hr/>					
MULTI_STAGE_HEURISTIC	871.02	967.07	2372.34	3.77	3.83
('ILS', 'None')	822.46	913.77	1438.66	53.11	55.08
('ILS', 'SA')	832.23	915.11	1536.80	53.75	55.44
('ILS', 'TS')	851.04	1113.80	9494.83	128.85	131.43
('ILS', 'ACO')	829.36	1154.03	13185.84	117.57	117.77
GA	828.19	867.63	7851.96	111.69	115.06
ALNS	838.48	915.78	1515.96	51.32	53.52
BB	906.08	-	-	1.12	1.13

:B-n45-k6





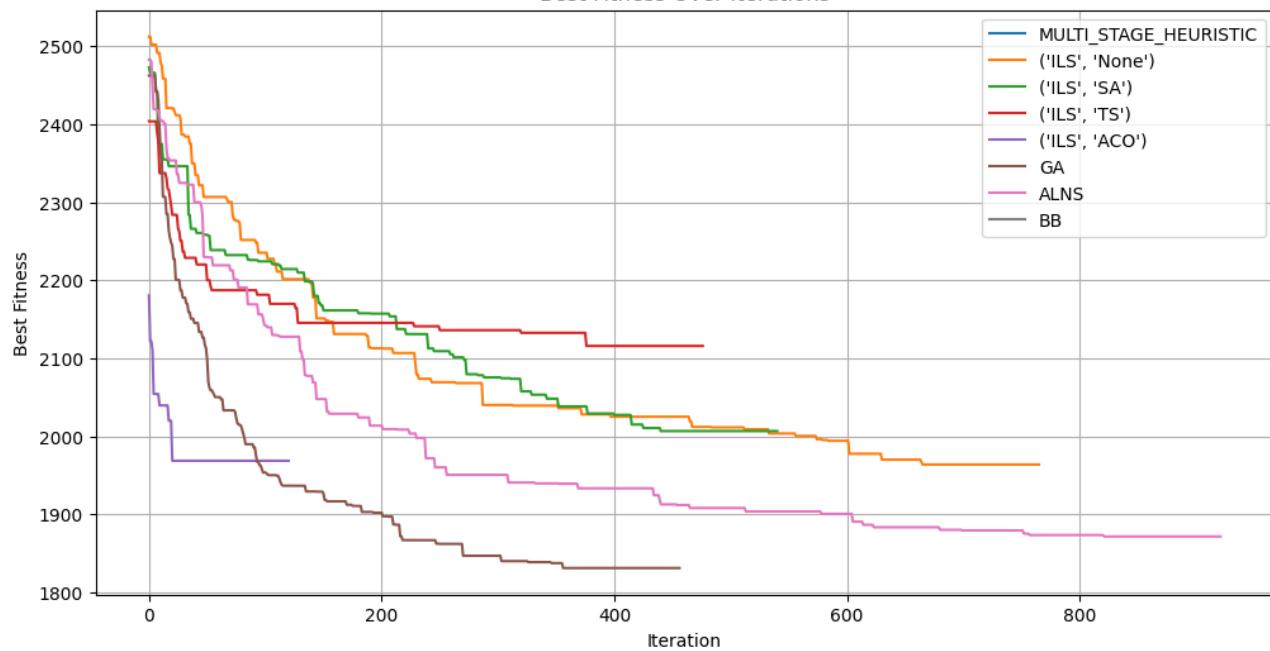




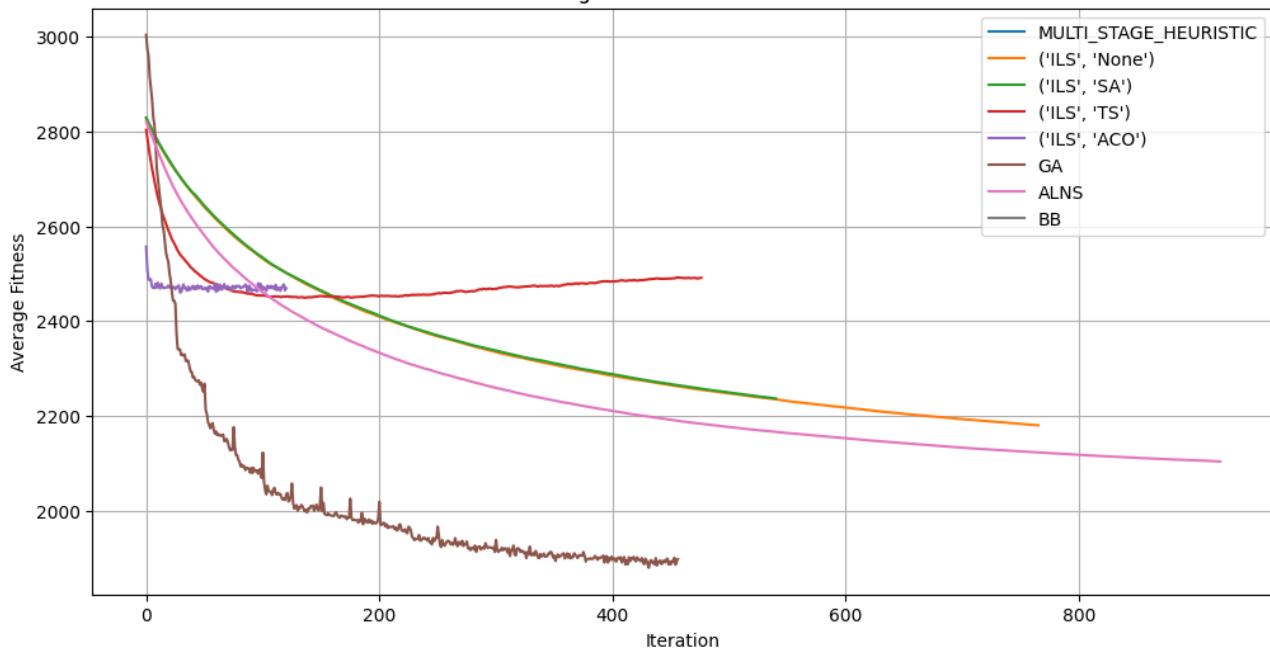
Algorithm	Best Fitness	Average Fitness	Variance	Elapsed Time	CPU Time
MULTI_STAGE_HEURISTIC	726.56	1176.34	30767.51	86.99	89.89
('ILS', 'None')	726.22	1126.85	24436.07	167.53	175.86
('ILS', 'SA')	746.63	1145.79	24070.25	153.99	163.03
('ILS', 'TS')	759.42	1312.73	48510.31	261.52	269.68
('ILS', 'ACO')	715.66	942.43	5898.85	500.09	506.90
GA	723.50	756.64	4647.03	280.99	300.00
ALNS	731.87	1132.50	26072.01	161.63	170.99
BB	781.36	-	-	5.25	5.76

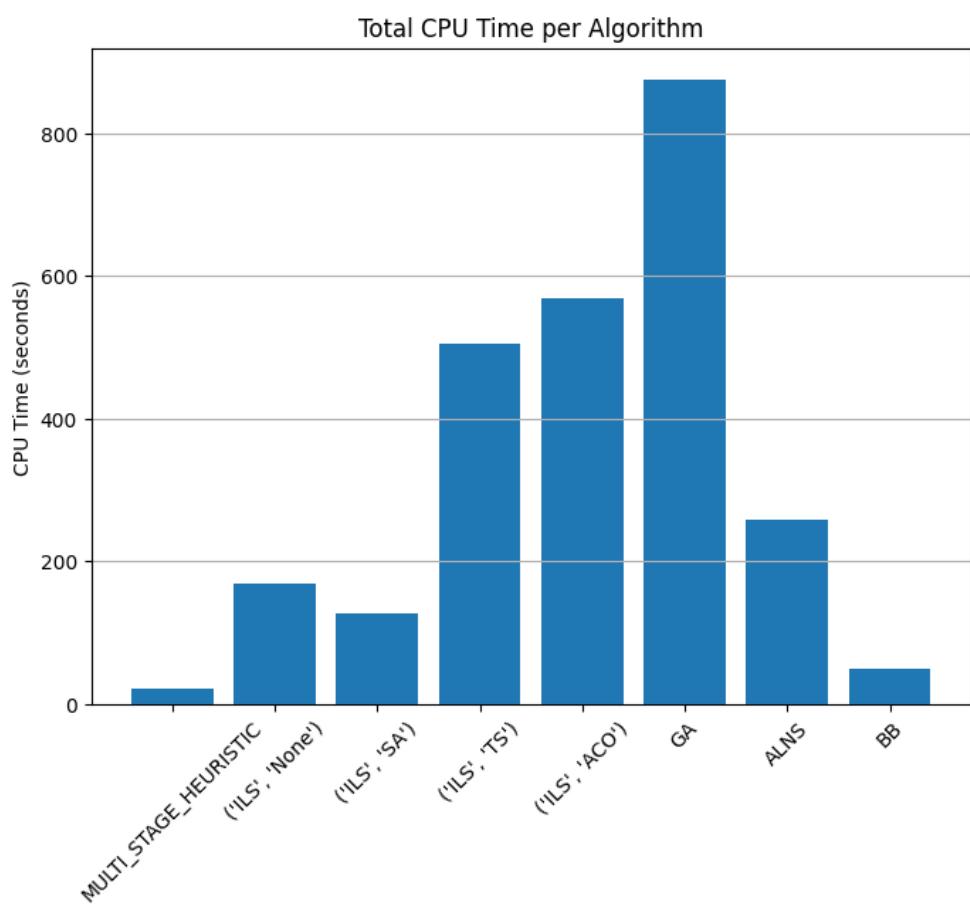
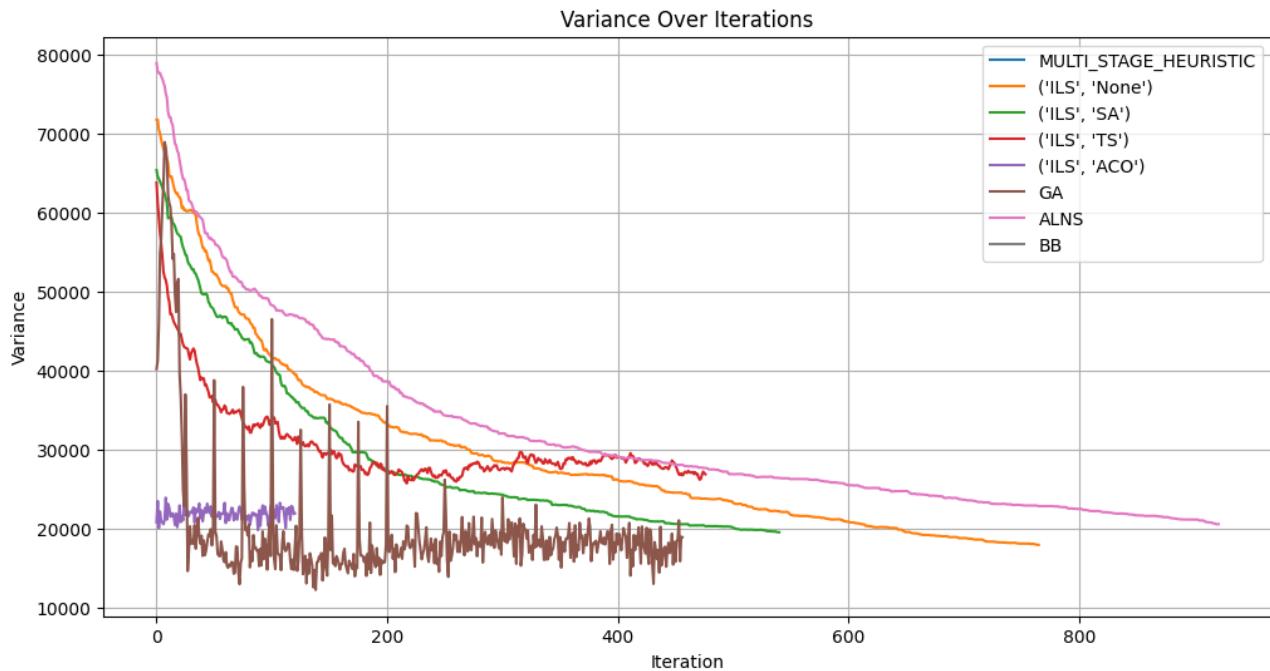
:A-n80-k10

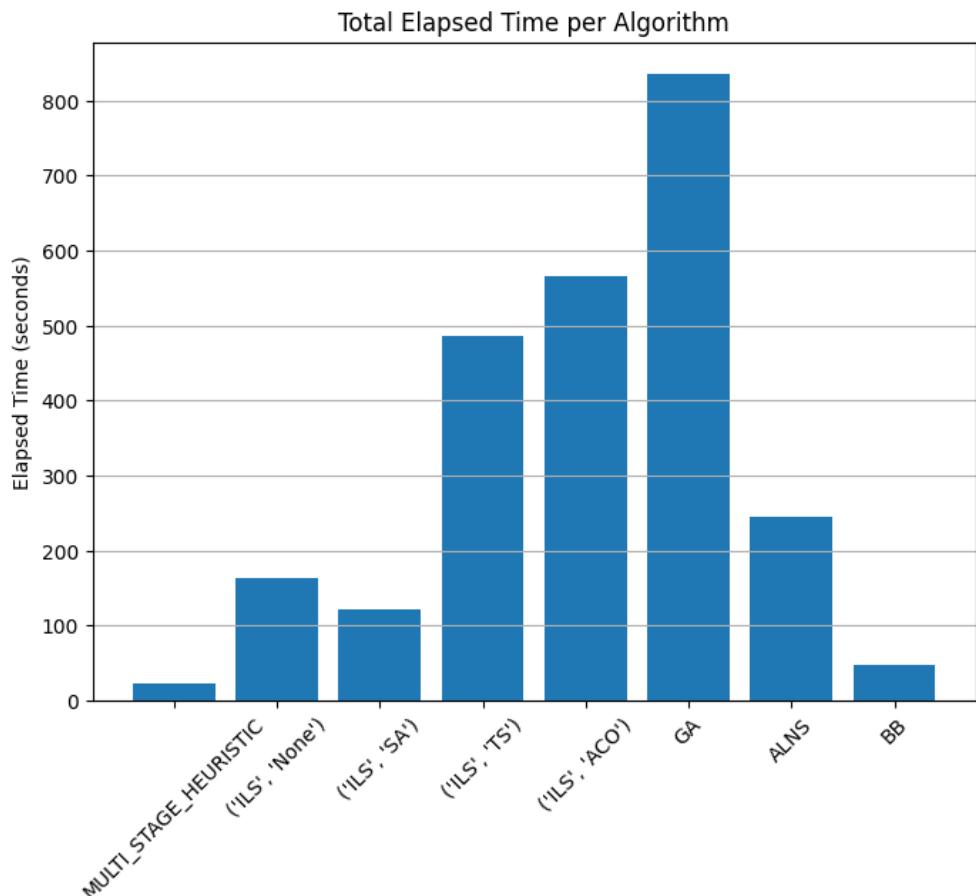
Best Fitness Over Iterations



Average Fitness Over Iterations

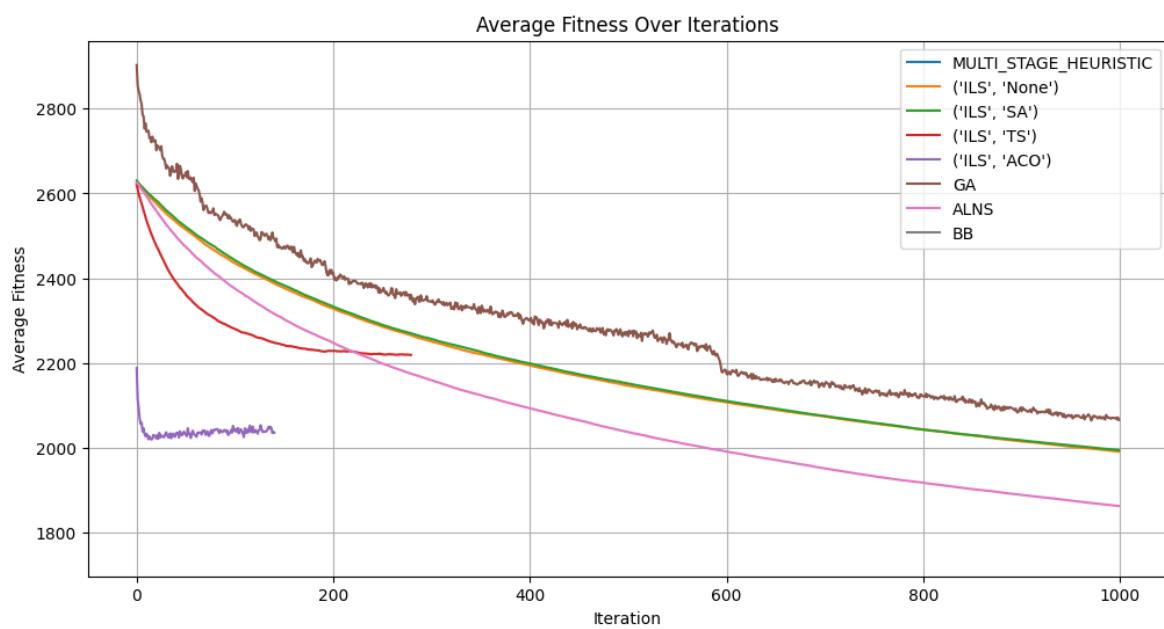
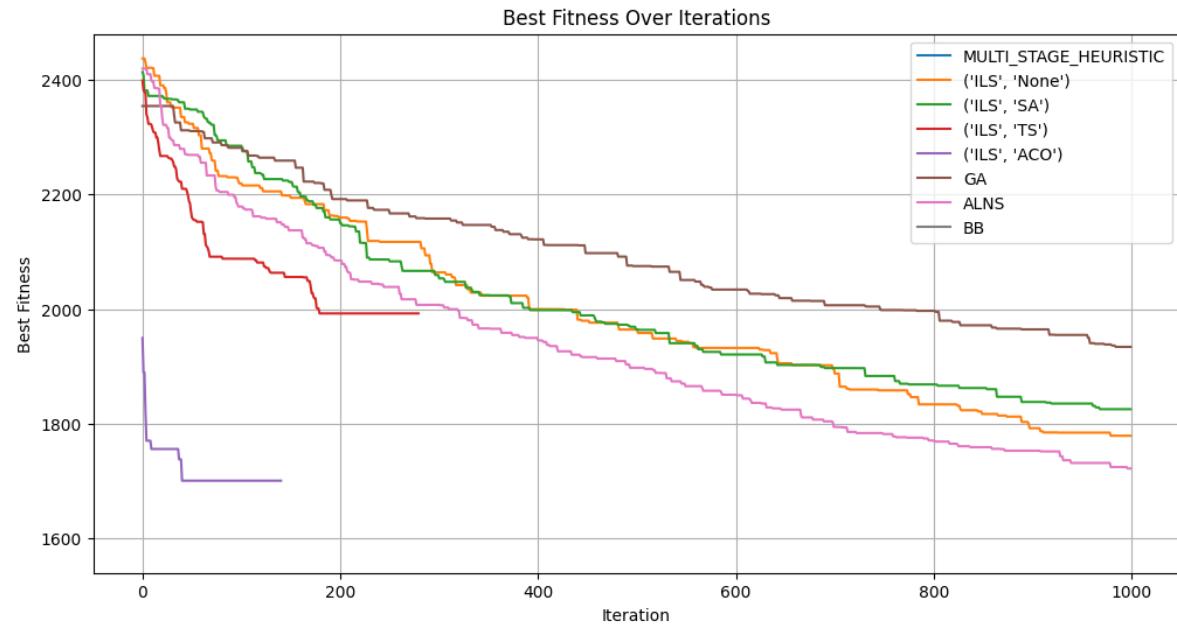


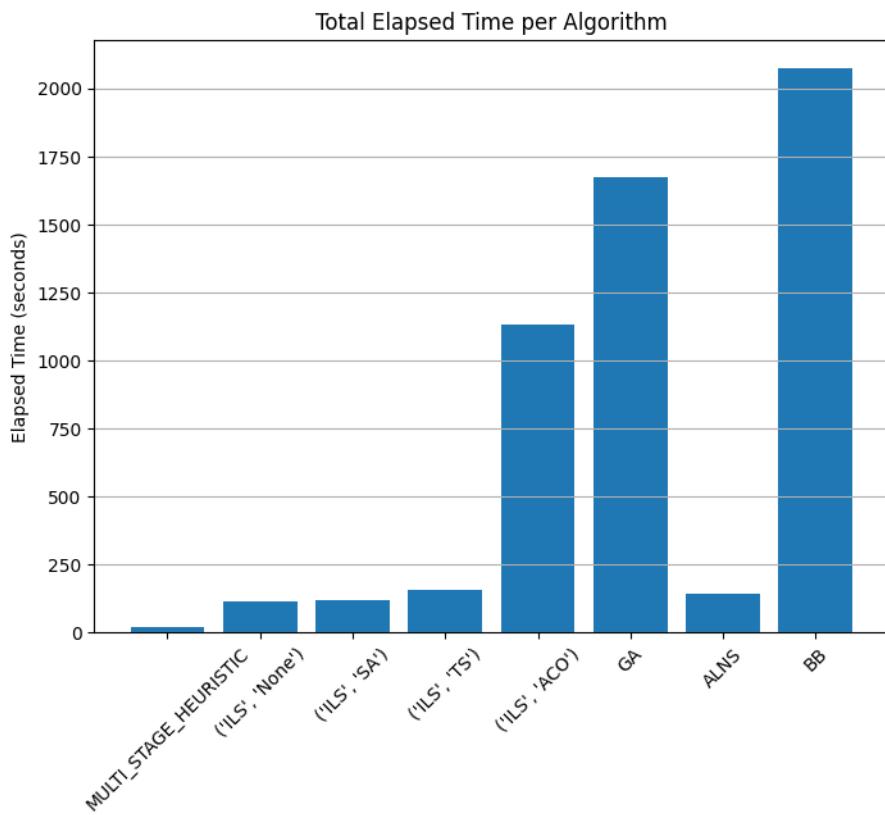
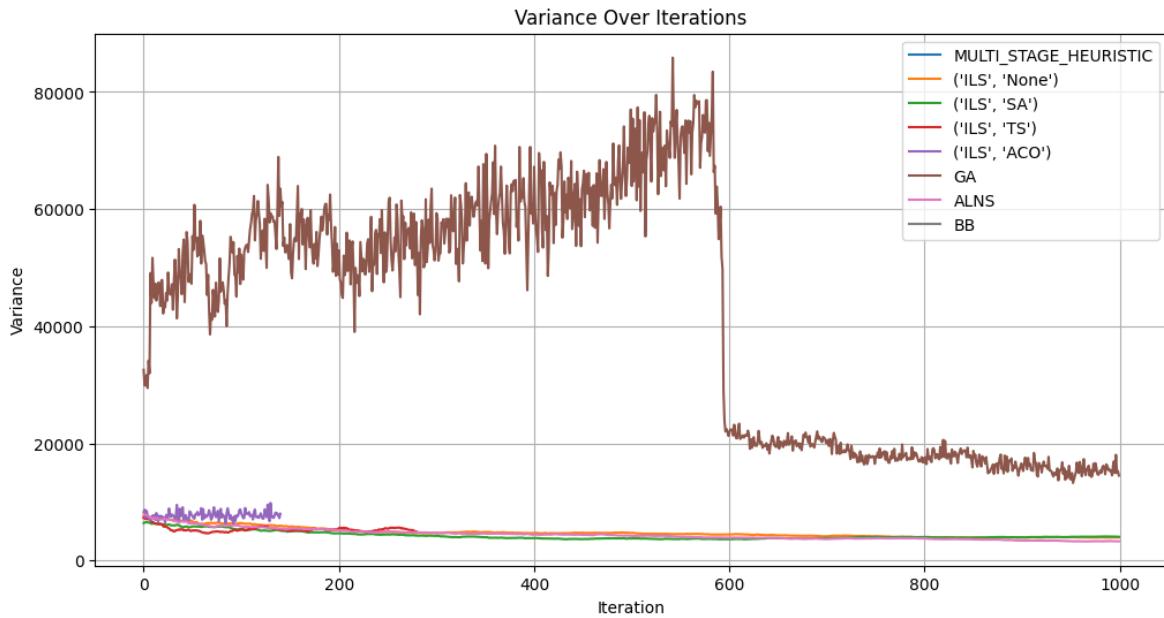


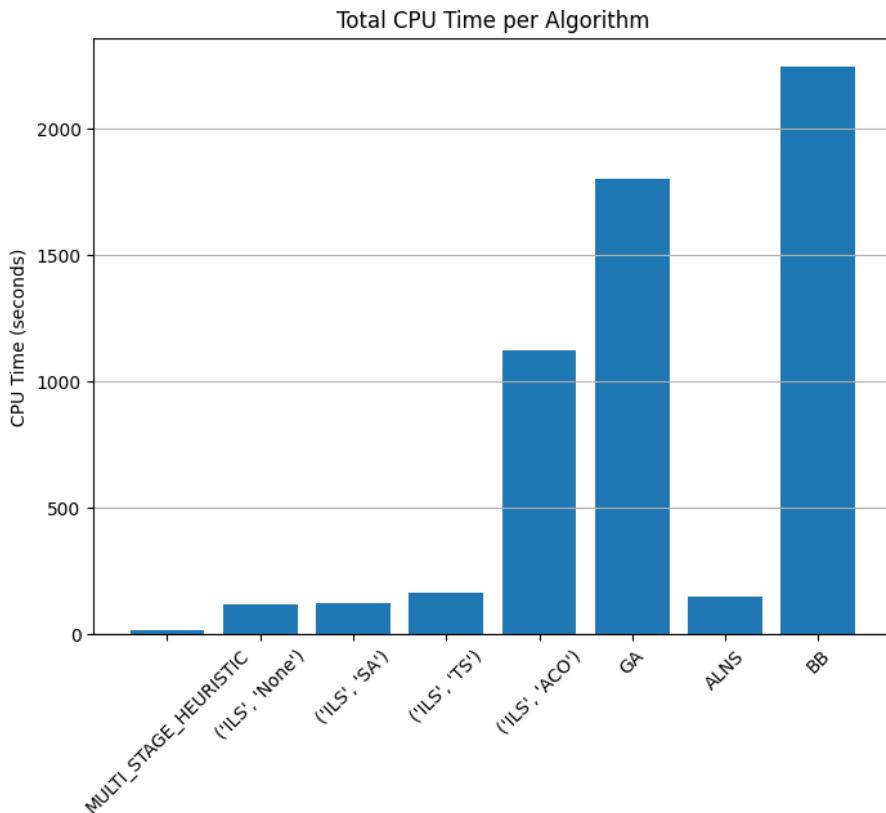


Algorithm	Best Fitness	Average Fitness	Variance	Elapsed Time	CPU Time
MULTI_STAGE_HEURISTIC	1962.96	2173.10	21282.96	22.27	21.90
('ILS', 'None')	1963.94	2179.98	17913.39	162.88	169.14
('ILS', 'SA')	2006.83	2236.41	19510.05	122.16	127.46
('ILS', 'TS')	2115.95	2491.19	26853.80	485.89	505.05
('ILS', 'ACO')	1968.76	2469.26	21896.79	566.53	568.74
GA	1831.30	1897.45	18885.13	836.00	875.29
ALNS	1871.71	2103.57	20539.82	244.59	258.15
BB	2048.17	-	-	46.56	49.24

:M-n200-k17







Algorithm	Best Fitness	Average Fitness Variance	Elapsed Time	CPU Time
MULTI_STAGE_HEURISTIC	1599.22	1753.84	3806.47	16.65
('ILS', 'None')	1779.30	1990.67	3942.68	111.72
('ILS', 'SA')	1825.62	1994.07	4035.81	115.19
('ILS', 'TS')	1992.55	2218.91	4960.63	152.77
('ILS', 'ACO')	1700.90	2035.58	7964.10	1129.55
GA	1934.27	2065.22	14494.34	1670.35
ALNS	1722.21	1862.23	3265.84	138.46
BB	1582.91	inf	inf	2073.80
				2247.93

ברוב המקרים אלגוריתם ה- Multi-Stage Heuristics היה החלש ביותר במונחים של הפיטניש הטוב ביותר והפיטניש הממוצע, אולם הפגין יכולות מבחינות סיבוכיות זמן. כמו כן, ניתן לשים לב לשונות הגבואה בין הפתרונותות שלו בתום ריצתו זהה ככל הנראה בשל אופיו הלא איטרטיבי ואי הכלתו למנגנון אינטראקציה בין הפתרונותות כמו במקרה האלגוריתם הגנטי ואלגוריתם ILS למשל. למרות שלא נדרש מאיתנו, בחרנו לכלול את גרסת ILS שלא משתמשת במיטה-היוריסטיקה כלשהי בשל כך שהבחנו בתוצאות התובות שהוא משיג. דבר זה נכיר יותר בבעיות קטנות ( מבחינת מספר הערים ומספר המסלולים ) לעומת בעיות בעלות סדר גדול, שם המיטה-היוריסטיקה מתגליה כבעל ערך רב בהיחלצות ממינימום לוקאלי. כמו כן, אפשר להבחן, באופן הגיוני, הקשר הופכי בין מספר האיטרציות והשונות, ככל שמספר

האייטרציות שהאלגוריתם דורש גדול יותר, השונות בין פתרונותיו קטנה. אלגוריתם ה- Branch and Bound דרש בד"כ זמן גדול יותר בהרבה מאשר האלגוריתמים, דבר הנובע מאופן המימוש שלו, זהה למורות השיפורים שבוצעו בו, בכך שהוא לוקח בחשבון את 3 השכנים הקרובים ביותר במקום את כל שאר הערים, ואת הגיזום שהוא עושה למצבים לא מבטיחים. אלגוריתם ה- ALNS הציג יכולת טובה בשדרוג הפתרון הטוב ביותר לאורך האיטרציות תוך שמירה על שונות מתונה, דבר המעיד על איזון טוב בין Exploration ל- Exploitation. אלגוריתם שהציג שונות גבוהה במיוחד היה האלגוריתם הגנטי, עדות להצלחתו של מנגנון האיים, אולם נראה שהוא בא על חשבון התוכנות מהירה ברוב המקרים.