Algoritmul Tabu Search pentru Problema Comis-Voiajor:

Algoritm:

Presupune păstrarea într-o memorie **tabu_list**, la fiecare iterație din cele **k**, a indiciilor luați în considerare de către algoritm pentru generarea următorului vecin și marcarea acestuia ca 'tabu' pentru un număr **r** de iterații.

Metode:

def **tabu_search_TSP**(dm: list, max_iter: int, r: int, search_type: str):

- Aplicarea algoritmului Tabu Search pentru problema TSP.
    1. Initializăm valorile dimension – dimensiunea matricii de distanțe, c - soluția curentă, inițial aleatoare, c_best și fitness_best și tabu_list – memoria tabu
    2. Executăm max_iter iterații
    3. În cadrul fiecărei iterații se caută cel mai bun vecin folosind *best_neighbour_TSP*
    4. La găsirea unui vecin cu un fitness mai slab, îl luăm și actualizăm best_fitness
    5. Actualizăm lista tabu folosind metoda *update_tabu_TSP*

def **best_neighbour_TSP**(c: list, dm: list, tabu_list: list, search_type: str):

- Găsește cel mai bun vecin pentru soluția actuală TSP.
    1. Initializăm valorile best_neighbour, best_neighbour_fitness și i_best, j_best
    2. Folosind 2 for-loops luăm perechi (i, j) de indici și aplicăm 2-swap sau 2-opt, în funcție de valoarea **search_type** pentru a găsi următorul vecin.
    3. Dacă găsim un vecin cu un fitness mai slab, în luăm și actualizăm i_best cu valoarea lui i, respectiv j_best cu valoarea lui j.

def **update_tabu_TSP**(tabu_list: list, i: int, j: int, r: int):

- Actualizează lista tabu pentru TSP prin marcarea ca 'tabu' a indiciilor i și j.
    1. Actualizăm pozitiile i și j ca fiind tabu pentru r iterații
    2. Decrementă restul pozitiilor pozitive

Algoritmul Tabu Search pentru Problema Rucsacului:

Algoritm:

Presupune păstrarea într-o memorie **tabu_list**, la fiecare iterație din cele **k**, a indiciilor luați în considerare de către algoritm pentru generarea următorului vecin și marcarea acestuia ca 'tabu' pentru un număr **r** de iterații.

Metode:

def **tabu_search_rucsac**(objects: list, max_capacity: int, max_iter: int, r: int):

- Aplicarea algoritmului Tabu Search pentru problema rucsacului.
    1. Initializăm valorile c - soluția curentă, inițial aleatoare, c_best și fitness_best și tabu_list – memoria tabu
    2. Executăm max_iter iterații
    3. În cadrul fiecărei iterații se caută cel mai bun vecin folosind *best_neighbour_rucsac*
    4. La găsirea unui vecin cu un fitness mai slab, îl luăm și actualizăm c_best și best_fitness
    5. Actualizăm lista tabu folosind metoda *update_tabu_rucsac*

def **best_neighbour_rucsac**(c: list, objects: list, max_capacity: int, tabu_list: list, fitness_best: int):

- Găsește cel mai bun vecin pentru soluția actuală a problemei rucsacului.
    1. Initializăm valorile x, fx și tabu_index
    2. Folosind 1 for-loops luăm i - indice și aplicăm bit-flip, modificand solutia curentă.
    3. Dacă găsim un vecin cu un fitness mai bun, în luăm și actualizăm tabu_index cu valoarea lui i.

def **update_tabu_rucsac**(tabu_list : list, index : int, r : int):

- Actualizează lista tabu pentru problem rucsacului prin marcarea ca 'tabu' a indicelui j si decrementarea celorlalti.
    1. Actualizăm pozitia i ca fiind tabu pentru r iterații
    2. Decrementă restul pozitiilor pozitive

Tabel cu rezultate pentru fiecare instanță a problemei prentru minim 5 valori diferite de parametri:

Analiza rezultatelor:

Comparație cu algoritmul HC pentru problema rucsacului:

Tabel cu valori pentru TS:

pi	n	k	r	Best Solution	Avg Solution	Avg Time
rucsac-20.txt	5	100	2	475	391.6	0.021039867401123048
rucsac-20.txt	5	100	5	587	497.6	0.020494604110717775
rucsac-20.txt	5	100	10	639	439.0	0.021574687957763673
rucsac-20.txt	5	100	20	462	429.2	0.034644031524658205
rucsac-200.txt	5	100	2	129606	120729.0	3.546601581573486
rucsac-200.txt	5	100	5	129718	124333.6	2.2254218101501464
rucsac-200.txt	5	100	10	124716	121231.2	1.9667951107025146
rucsac-200.txt	5	100	20	129150	125380.8	1.7487192153930664

Tabel cu valori pentru NAHC:

Problem Instance	n	k	Best Solution	Avg Solution	Avg Time
rucsac-20.txt	10	50	574	528.3	7.624626159667969e-05
rucsac-20.txt	10	100	591	513.5	0.00016393661499023438
rucsac-20.txt	10	200	694	532.4	0.0003656148910522461
rucsac-200.txt	10	50	133195	131960.3	0.0009571552276611328
rucsac-200.txt	10	100	132710	132009.4	0.0015289068222045898
rucsac-200.txt	10	200	132806	131980.7	0.0026607275009155273

Comparație atât pentru SA, cât și pentru TS:

pi	n	k	T	t_min	alpha	Best Solution	Avg Solution	Avg Time
rucsac-20.txt	5	10	100	0.001	0.6	655	569.2	0.0009380817413330078
rucsac-20.txt	5	10	100	0.001	0.999	726	703.0	0.5281653881072998
rucsac-20.txt	5	10	100	1e-06	0.6	653	607.0	0.005172872543334961
rucsac-20.txt	5	10	100	1e-06	0.999	726	693.2	0.7094807624816895
rucsac-20.txt	5	10	1000	0.001	0.6	650	530.4	0.0015798091888427734
rucsac-20.txt	5	10	1000	0.001	0.999	718	699.2	0.6403846740722656
rucsac-20.txt	5	10	1000	1e-06	0.6	623	564.4	0.0023781776428222655
rucsac-20.txt	5	10	1000	1e-06	0.999	726	702.8	0.6774191856384277
rucsac-200.txt	5	10	100	0.001	0.6	133351	132429.8	0.00351715087890625
rucsac-200.txt	5	10	100	0.001	0.999	132924	132486.0	2.165087890625
rucsac-200.txt	5	10	100	1e-06	0.6	132138	131838.0	0.005373668670654297
rucsac-200.txt	5	10	100	1e-06	0.999	132632	132168.8	2.988794183731079
rucsac-200.txt	5	10	1000	0.001	0.6	132540	132072.2	0.004440927505493164
rucsac-200.txt	5	10	1000	0.001	0.999	132922	132646.0	2.385439729690552
rucsac-200.txt	5	10	1000	1e-06	0.6	132574	132180.2	0.006432533264160156
rucsac-200.txt	5	10	1000	1e-06	0.999	133217	132706.8	3.854534959793091

pi	n	k	r	Best Solution	Avg Solution	Avg Time
rucsac-20.txt	5	100	2	475	391.6	0.021039867401123048
rucsac-20.txt	5	100	5	587	497.6	0.020494604110717775
rucsac-20.txt	5	100	10	639	439.0	0.021574687957763673
rucsac-20.txt	5	100	20	462	429.2	0.034644031524658205
rucsac-200.txt	5	100	2	129606	120729.0	3.546601581573486
rucsac-200.txt	5	100	5	129718	124333.6	2.2254218101501464
rucsac-200.txt	5	100	10	124716	121231.2	1.9667951107025146
rucsac-200.txt	5	100	20	129150	125380.8	1.7487192153930664

Comparație între SA și TS pentru problema TSP:

pi	n	k	T	t_min	alpha	Best Solution	Avg Solution	Avg Time
pr107.tsp	5	10	100	0.001	0.6	600921	576541.8	0.003697490692138672
pr107.tsp	5	10	100	0.001	0.999	580935	560493.2	1.8919325351715088
pr107.tsp	5	10	100	1e-06	0.6	586600	561025.4	0.005951070785522461
pr107.tsp	5	10	100	1e-06	0.999	636709	586694.6	3.3370370864868164
pr107.tsp	5	10	1000	0.001	0.6	605177	579364.0	0.004155254364013672
pr107.tsp	5	10	1000	0.001	0.999	603705	572750.4	2.3923779010772703
pr107.tsp	5	10	1000	1e-06	0.6	643257	574505.8	0.006250953674316407
pr107.tsp	5	10	1000	1e-06	0.999	582434	570570.6	3.5796385288238524

pi	n	k	r	search_type	Best Solution	Avg Solution	Avg Time
pr107.tsp	5	10	2	2-opt	353793	397795.0	2.6277873516082764
pr107.tsp	5	10	5	2-opt	358961	394455.2	2.374435043334961
pr107.tsp	5	10	10	2-opt	386258	409556.6	2.3648256778717043
pr107.tsp	5	10	20	2-opt	351525	389925.0	2.3527758598327635
pr107.tsp	5	10	2	2-swap	305971	330952.8	2.277150869369507
pr107.tsp	5	10	5	2-swap	313008	331921.6	2.3883225440979006
pr107.tsp	5	10	10	2-swap	307649	338973.4	2.41811146736145
pr107.tsp	5	10	20	2-swap	297086	333107.2	2.395489311218262

Observăm astfel că pentru un număr k mai mic de iterații, algoritmul Tabu Search a reușit obtinea unui rezultat mai slab decat Next Ascent Hill Climbing în cazul Problemei Rucsacului prin marcarea acelor biți luați în calculul soluției ca fiind tabu pentru un număr r variat de iteratii, cea mai buna soluție fiind obținută pentru r = 10, care a fost totuși mai slabă decât cea obținută de NAHC. Cu toate acestea, se observă că algoritmul Simulated Anealling a obținut rezultate mai bune decat TS sau NAHC prin elementul probabilistic introdus, acest algoritm reușind să nu se blocheze pe minime/maxime locale mai bine decat TS sau NAHC. Același lucru nu este valabil și pentru Problema Comis-Voiajorului pentru care SA a obținut un rezultat mult mai slab decât TS.