УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА У НОВОМ САДУ

Марко Николић

Тура кроз европске градове

ПРОЈЕКТНИ ЗАДАТАК

Нови Сад, 2025.

САДРЖАЈ

1. МОТИВАЦИЈА И ИЗБОР АЛГОРИТМА

Проблем планирања туре кроз европске градове представља пример проблема путујућег трговца (Travelling Salesman Problem – TSP). Овај проблем је веома комплексан јер број могућих тура расте факторијелно са бројем градова. Приступи попут backtracking алгоритма имају временску сложеност O(n!), што их чини непрактичним за веће скупове података.

Да би се смањила временска комплексност, потребно је применити оптимизације које омогућавају елиминацију неперспективних решења у раној фази. Један од начина је да се препознају стања која су већ посећена уз мањи трошак, те да се такве туре одбаце.

Да би се постигла оптимизација и елиминација неперспективних тура, изабран је алгоритам који комбинује енумерацију градова, представљање посећених стања бит-маскама и динамичко програмирање.

2. ОПИС АЛГОРИТМА

2.1 Учитавање података и енумерација градова

Први корак алгоритма је учитавање градова и растојања између њих из фајла. Градови су представљени као низови знакова (string), па би за чување међусобних растојања могла да се користи хеш мапа. Међутим, због често потребног приступа растојањима током извршавања алгоритма, коришћење хеш мапе постаје скупо, јер свака претрага подразумева израчунавање хеша. Због тога је неопходно оптимизовати начин складиштења и приступа растојањима између градова.

За решавање овог проблема употребљена је енумерација градова. Сваком граду је додељен број од 0 до n – 1. У хеш мапи се чувају парови кључ–вредност (слика 1), где је кључ назив града (string), а вредност индекс од 0 до n – 1, који представља град у току извршавања алгоритма. Такође, у једном вектору се чувају називи градова (string) поређани по додељеном броју (слика 2). Тај вектор служи за реконструкцију најкраће туре како би могла да се испише у читљивом облику.

Key	Value
Paris	0
London	1
Lisbon	2
Kyiv	3

Слика 1 - Хеш мапа (град – индекс)

0	1	2	3
Paris	London	Lisbon	Kyiv

Слика 2 - Вектор назива градова

Након енумерације градова, хеш мапа за чување растојања више није потребна. Уместо ње користи се листа суседства (слика 3), односно матрица величине n x n, где ћелија mat[i][j] представља растојање између градова i и j. Листа суседства се креира уз помоћ хеш мапе која мапира називе градова на њихове индексе. Приступ растојањима у листи суседства се обавља директно помоћу индекса, без потребе за хеширањем као код хеш мапе. Подаци се налазе у једном континуираном блоку, што повећава број кеш погодака и убрзава приступ.

	0	1	2	3
0	0	100	250	300
1	100	0	50	500
2	250	50	0	1000
3	300	500	1000	0

Слика 3 - Матрица суседства

2.2 Бит-маске стања и матрица стања

У току алгоритма прате се стања у којима тура може да се нађе, како би се могле одбацити туре које долазе до неког стања уз већи трошак од друге туре која пролази кроз исто стање. Стање идентификују две информације:

1. Листа посећених градова,
2. Последњи посећени град.

За свако стање потребно је пратити 2 чињенице:

1. Најкраћи прећени пут који води то тог стања,
2. Претходни град из туре са најкраћим прећеним путем (потребно за реконструкцију туре).

Погодан начин за чување података о стањима је матрица чије димензије представљају листу посећених градова и последњи посећени град, а вредности у ћелијама чувају најкраћи прећени пут и претходни град из туре која га је довела до тог стања.

Сваки град може бити или посећен или непосећен, што значи да постоји 2^n могућих комбинација посећених градова. Чување свих тих листи је меморијски неефикасно, па се овај проблем заобилази коришћењем бит-маски, где сваки бит представља информацију о посећености једног града према претходној енумерацији градова. На пример, ако је трећи бит у маски 1, трећи град је већ посећен, док 0 означава да није посећен. На овај начин се информације о посећености свих градова чувају у једном целобројном типу (int), што је знатно ефикасније него коришћење листи. Имајући ово у виду, прва димензија матрице је 2^n.

Друга димензија матрице представља последњи посећени град. Пошто постоји *n* градова, друга димензија матрице је величине *n*.

Дакле, матрица стања има димензије $2^n \times n$ (слика 4). У току алгоритма се ова матрица попуњава све док се не израчуна најкраћи пут за долазак у стање 0b111...1.

	0	1	2	3
0b0000				
0b0001				
0b0010				
0b0011				
0b0100				
0b0101				
0b0110				
0b0111				
0b1000				
0b1001				
0b1010				
0b1011				
0b1100				
0b1101				
0b1110				
0b1111				

Слика 4 - Матрица стања

2.3 Проналажење најкраће туре (динамичко програмирање)

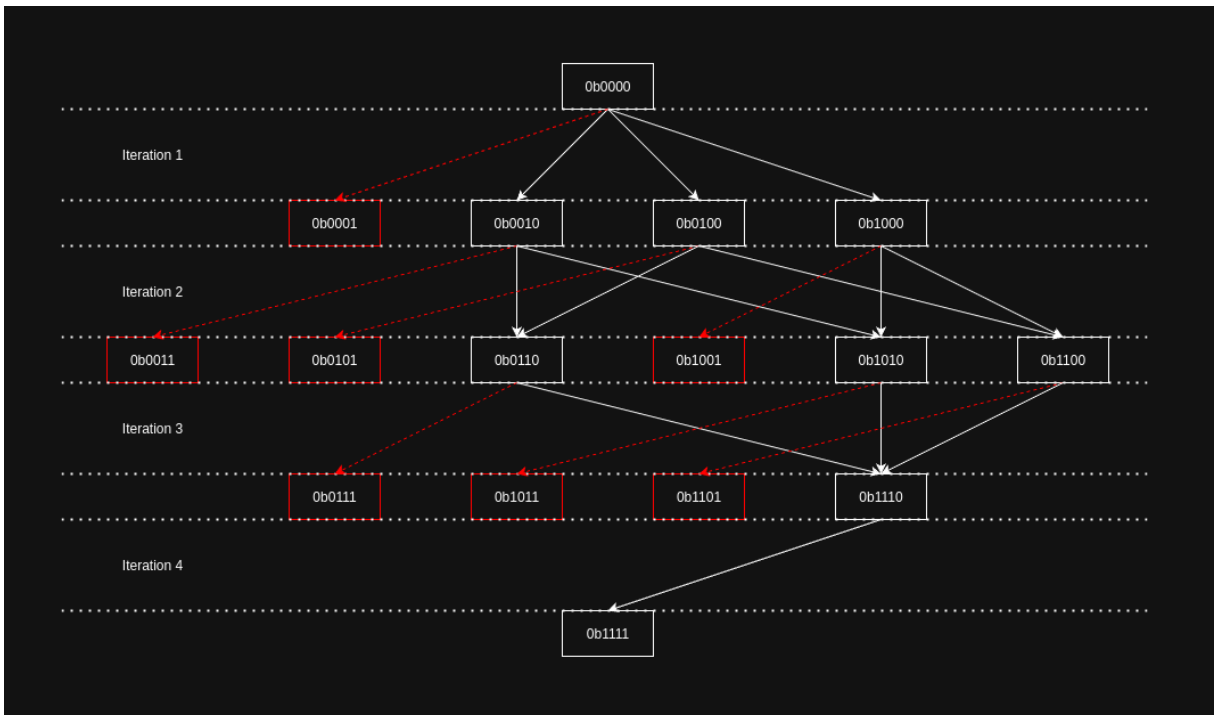
За попуњавање матрице стања погодан је приступ динамичког програмирања. Овај приступ користи итерације, тако да се проблем за n градова решава постепено: прво за стања са једним посећеним градом, затим са два, и тако даље, све док се не обухвате сви градови. Ово значи да алгоритам има n итерација.

У контексту проблема путујућег трговца, динамичко програмирање почиње од стања у којима није посећен ниједан град (стање у ком се налазимо у првом граду у који се на крају враћамо). У првој итерацији израчунавају се најкраћи путеви за сва стања у

којима је посећен тачно један град. У следећој итерацији као улаз служе већ израчуната стања са једним посећеним градом, што омогућава израчунавање најкраћих путева за стања у којима су посећена тачно два града. Процес се понавља све док се не обраде стања са свим градовима, чиме се у матрици стања налази најкраћи пут за сваки могући подскуп посећених градова и сваки последњи посећени град. Крајњи град се заобилази до последње итерације јер је по спецификацији проблема он први и последњи у тури. То значи да се сва стања у којима је почетни/крајњи град посећен одбацују, осим стања у ком су сви градови посећени.

Током итерација може се десити да се исто стање израчуна више пута, али у матрицу стања се уписује само најкраћи пут (и претходни град из тог пута), док се сви остали резултати одбацују. На овај начин се избегава задржавање неперспективних тура и значајно смањује број потребних израчунавања у односу на наивно обилажење свих тура.

На слици 5 поједностављено су приказане итерације алгоритма на примеру n = 4. Дијаграм је поједностављен тако што не приказује последњи посећени град у сваком стању, већ само бит-маску посећених градова. У суштини, свако стање на дијаграму одговара једном реду у матрици стања.



Слика 5 - Итерације алгоритма

2.4 Реконструкција најкраће туре

На крају алгоритма, матрица стања је потпуно попуњена. Дужина најкраћег пута налази се у ћелији mat[2^n – 1][0], која представља стање у ком су посећени сви градови, а последњи посећени град је град са индексом 0, односно почетни/крајњи град.

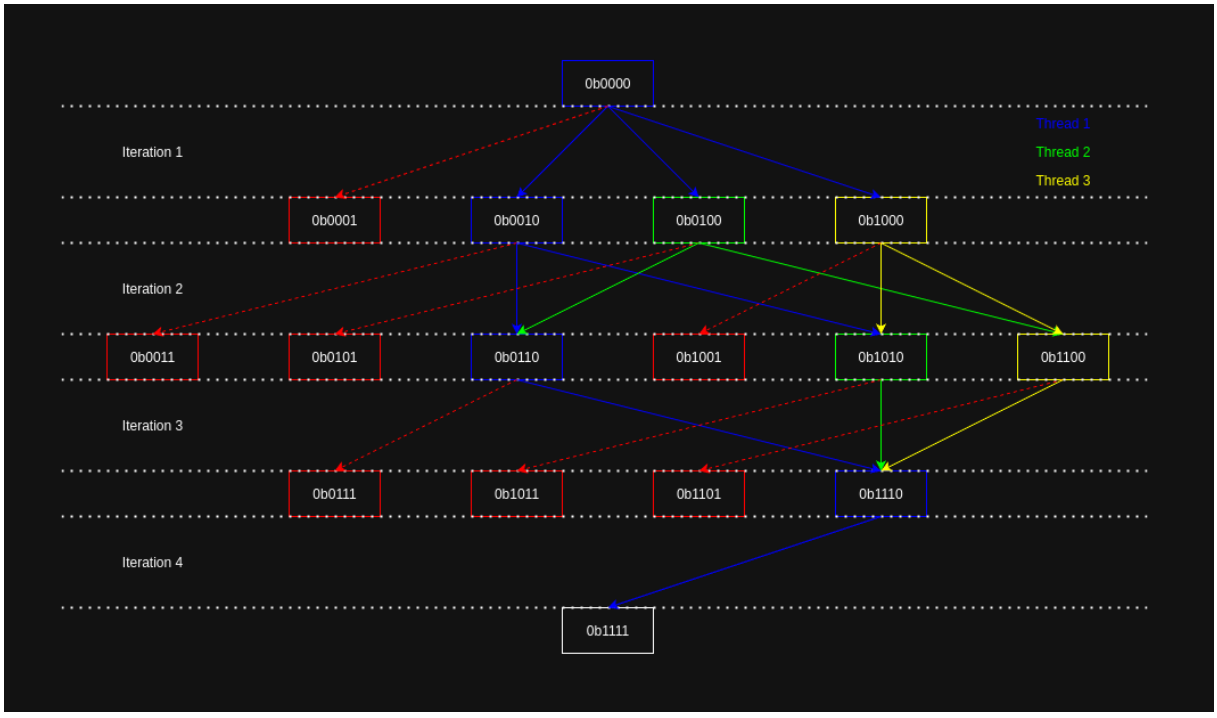
Поред саме дужине најкраће туре, потребно је реконструисати редослед градова у тури. Градови су нумерисани бројевима од 0 до n – 1, а њихови називи се чувају у вектору (појашњеном на слици 2), где је сваки индекс мапиран на одговарајући град. Реконструкција туре врши се уз помоћ информација о претходном граду које се чувају у

матрици стања. Процес се обавља уназад: креће се од последњег стања (где су посећени сви градови) и пратећи претходне градове, корак по корак се реконструише цела тура све до почетног/крајњег града. На овај начин добија се редослед градова кроз које пролази најкраћа могућа тура.

2.5 Паралелизација

Да би се алгоритам додатно убрзао, потребно је применити паралелизацију. Природна идеја је паралелизовати итерације динамичког програмирања, али то није могуће јер се оне морају извршавати редом, излаз једне итерације служи као улаз у наредну.

Међутим, паралелизација је могућа унутар сваке итерације. Улаз у једну итерацију представља скуп стања у којима је посећено тачно і градова (где је і број текуће итерације). Из ових стања израчунавају се најкраћи путеви за стања у којима је посећено і + 1 градова. У секвенцијалној варијанти алгоритма ова обрада се врши редом, али се она може паралелизовати јер свака транзиција може бити израчуната независно од осталих (слика 6).



Слика 6 - Итерације паралелног алгоритма

Изазов при оваквој паралелизацији је упис у матрицу стања. Већ је истакнуто да се исто стање у оквиру једне итерације може израчунати више пута, а у матрицу се уписује само најкраћи пут. Због тога је матрица стања дељени ресурс, и паралелни уписи у исту ћелију могу довести до губитка оптималне вредности. Да би се то спречило, у паралелној верзији алгоритма свака ћелија матрице стања има придружен mutex који се закључава при сваком упису.

2.6 Временска и просторна комплексност

Временска комплексност алгоритма: $O(2^n \times n^2)$. Постоји $2^n \times n$ могућих стања кроз које алгоритам пролази. При израчунавању сваког новог стања, разматрају се сви могући прелази (до n градова), што даје додатни фактор n.

Просторна комплексност алгоритма: $O(2^n \times n)$. Матрица стања има димензије $2^n \times n$.

2.7 Уска грла

При паралелизацији алгоритма, један од кључних изазова је синхронизација и судар при упису. Свака ћелија матрице стања представља дељени ресурс, па се при упису најкраћег пута користе мутекси како би се спречио губитак података. Ово закључавање може узроковати чекање нити, што смањује паралелизам и доводи до мањег убрзања од теоретски могућег.

Због великог обима података ($2^n \times n$ матрица), приступи меморији могу утицати на перформансе. Смањење локалитета и кеш-погодака резултује додатним кашњењима.

У смислу меморијских захтева, матрица стања је великог обима за веће n, што ограничење меморије чини значајним фактором, нарочито када се додају паралелни уписи и потреба за закључавањем.

Иако паралелизација убрзава извршавање, алгоритам је и даље временски сложен ($O(2^n \times n^2)$), па за веће n време извршавања расте експоненцијално и паралелизација само делимично ублажава проблем. Ово значи да, и поред убрзања, за веће скупове градова алгоритам постаје ограничен и потребне су додатне оптимизације или промена алгоритма.

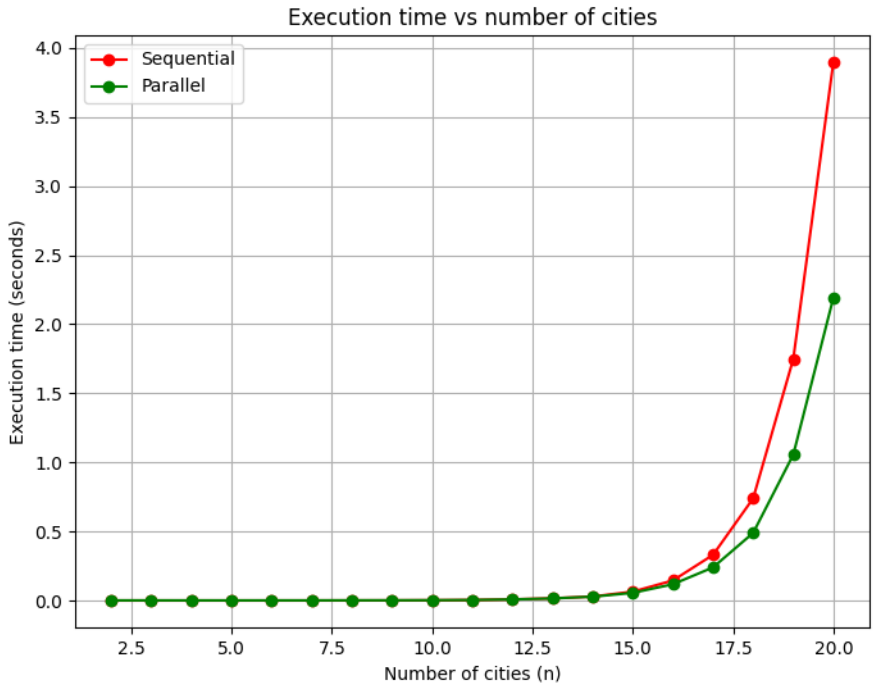
3. МЕТРИКА ПЕРФОРМАНСИ

Алгоритам је тестиран за све вредности n од 2 до 20. За сваки избор n, извршено је мерење времена извршавања како би се добила процена перформанси (слика 7). Паралелна верзија алгоритма показује убрзање у односу на секвенцијалну имплементацију, а уочено је да убрзање расте са повећањем величине проблема, односно са порастом вредности n. Ово је у складу са очекивањима јер већи проблеми омогућавају бољу искоришћеност доступних процесорских ресурса.

n	Sequential execution time	Parallel execution time	Speedup
2	0.000006	0.00075	0.008
3	0.000007	0.000881	0.007945
4	0.00001	0.000851	0.011751
5	0.00002	0.000783	0.025542
6	0.000043	0.000888	0.048423
7	0.000103	0.000924	0.111471
8	0.000251	0.001335	0.188015
9	0.000614	0.001527	0.402096
10	0.00143	0.00212	0.674528
11	0.003414	0.00376	0.907979
12	0.007903	0.007773	1.016725
13	0.015833	0.015067	1.050839
14	0.028672	0.027692	1.035389
15	0.064035	0.055341	1.157099
16	0.144954	0.117442	1.234261
17	0.330223	0.239558	1.378468
18	0.739357	0.489249	1.511208
19	1.74805	1.055424	1.656254
20	3.897250	2.191662	1.778217

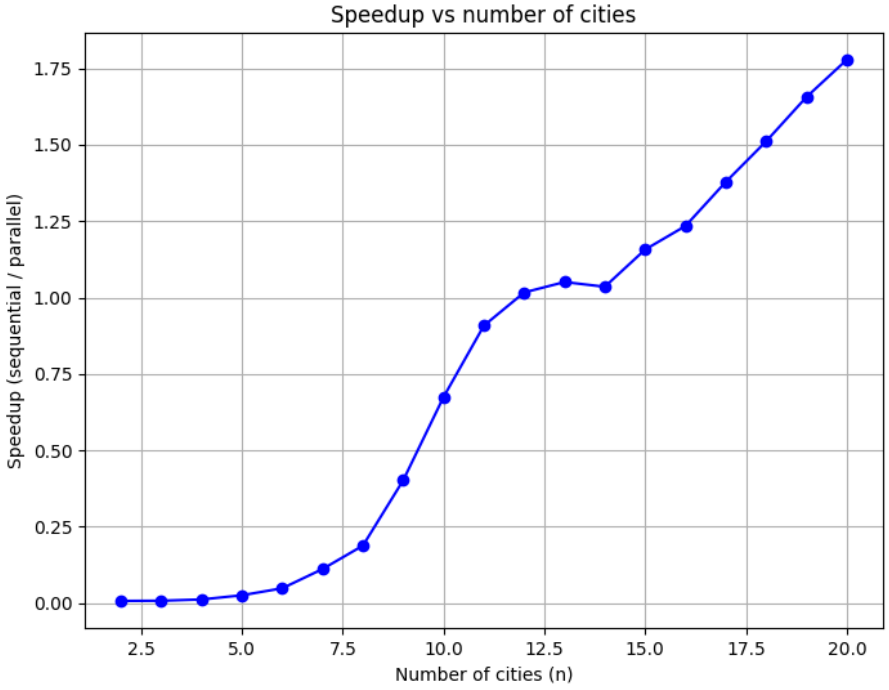
Слика 7 - Резултати мерења

На графику на слици 8 су приказана времена извршавања алгоритма у зависности од n.



Слика 8 - График времена извршавања

На графику на слици 9 су приказана убразања пararелног алгоритма у односу на секвенцијални алгоритам у зависности од n.



Слика 9 - График убрзања