



УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА
У НОВОМ САДУ



Паралелизација претраге у ширину и A^* алгоритма

Семинарски рад
- МАСТЕР АКАДЕМСКЕ СТУДИЈЕ -

Студент:

Огњен Кузановић

E2 33/2021

Нови Сад, 2022. године

Сажетак

Предмет овог рада била је паралелизација претраге у ширину заједно са A^* алгоритмима који представљају једне од најпознатијих алгоритама претраге над граф структурама података. Паралелизација је вршена над граф структуром података која је креирана на основу матрице поља где је циљ био наћи путању којом се може доћи од почетног чвора до крајњег чвора заобилазећи препреке тј. поља по којима се не може кретати. У свим случајевима тестирања, паралелизација је довела до успорења рада претраге код оба алгоритма.

САДРЖАЈ:

1. Увод	1
2. Креирање графа	2
3. Претрага у ширину	6
3.1 Имплементација паралелне претраге у ширину	8
3.2 Поређење перформанси	12
4. А* алгоритам	14
4.1 Имплементација паралелног А* алгоритма	15
4.2 Поређење перформанси	18
5. Закључак	19
Литература	20

1. УВОД

У овом раду представљен је један приступ паралелизацији претраге у ширину и A* алгоритма као једних од најпознатијих алгоритама претраге над граф структуром података који су нашли примену у многим областима. Пре саме паралелизације изгенерисана је граф структура података на основу матрице поља где је идеја да се пронађе путања којом се може стићи од почетног до крајњег чвора заобилазећи препреке тј. поља по којима је кретање омогућено. Паралелизација се најпре односи на делове алгоритама који су задужени за додавање нових чворова у ФИФО или приоритетне редове, а који ће у наредним корацима представљати ужи избор за следећу обраду.

За имплементационе потребе коришћен је програмски језик C заједно са *OpenMP* АПИ-јем помоћу којег се аотирањем кода одређеним директивама може остварити извршавање програма паралелно на више процесорских језгара.

У другом поглављу је дат увид у начин на који је креирана граф структура података на основу генерисане матрице поља. У трећем поглављу дат је преглед претраге у ширину заједно са паралелним приступом паралелизацији и извршено је поређење перформанси између секвенцијалног и паралелног приступа. У наредном, четвртом, поглављу одрађен је идентичан скуп ствари само на A* алгоритму, а у последњем поглављу дати су закључци до којих је аутор дошао приликом рада на овом раду.

2. КРЕИРАЊЕ ГРАФА

Како су граф структуре података у сржи претраге у ширину и A* алгоритма, било је потребно осмислити ефикасан начин за њихово креирање и визуализацију. Идеја је да се прво изгенерише матрица поља/ћелија која ће потом бити претворена у граф. Та матрица поља ће садржати препреке тј. ћелије по којима кретање није омогућено и потребно је стићи од почетног до крајњег поља користећи неки од претходно поменутих 2 алгоритма претраге.

Програмска имплементација ове матрице остварена је помоћу C програмског језика и фајла *generator.c*. Најбитнија функција овог фајла јесте функција *generateGraph* (Изворни код 1) која помоћу двоструке угњеждене *for* петље креира све ћелије матрице и обележава их са вредностима бројача петљи *i* и *j* односно редом и колоном у којима се тренутна ћелија налази, као и са ознаком *special* која сугерише чињеницу да ли је дато поље препрека или није. У том делу кода, може се навести услов под којим ће неко поље бити препрека. Након што су поља иницијално креирана, за свако поље се гледа да ли су поља око њега унутар граница матрице (Линија 29 у изворном коду 1 и цела функција у узворном коду 2) и ако јесу, она се додају као комшије поља што ће у контексту граф структуре података произвести ивице чворова.

```

1. struct Graph* generateGraph(){
2.     cells = (struct Cell**) malloc(ROWS * COLS * sizeof(struct
   Cell*));
3.
4.     for(int i = 0; i < ROWS; i++){
5.         for(int j = 0; j < COLS; j++){
6.             struct Cell* cell = (struct Cell*)
   malloc(sizeof(struct Cell));
7.             cell->i = i;
8.             cell->j = j;
9.             if((i == 2 && j < 5) || ((i == 5) && (j > 3)))
10.                 cell->special = true;
11.             else
12.                 cell->special = false;
13.
14.             cells[i * COLS + j] = cell;
15.         }
16.     }
17.

```

```
18.     struct FullEdge** edges = (struct FullEdge**) malloc(
19.     ROWS * COLS * 9 * sizeof(struct FullEdge*)
20.     );
21.     int numOfEdges = 0;
22.
23.     int numOfNodes = ROWS * COLS;
24.
25.     float* heuristics = (float*) malloc(numOfNodes *
26.     sizeof(float));
27.
28.     for(int i = 0; i < ROWS * COLS; i++){
29.         heuristics[i] = calculateHeuristics(cells[i]->i,
30.         cells[i]->j);
31.         addNeighbourCells(cells, i, edges, &numOfEdges);
32.     }
33.
34.     struct FullEdge* finalEdges = (struct FullEdge*)
35.     malloc(numOfEdges * sizeof(struct FullEdge));
36.
37.     for(int i = 0; i < numOfEdges; i++){
38.         struct FullEdge* edge = (struct FullEdge*)
39.         malloc(sizeof(struct FullEdge));
40.         edge->src = edges[i]->src;
41.         edge->dest = edges[i]->dest;
42.         edge->cost = edges[i]->cost;
43.
44.         finalEdges[i] = *edge;
45.     }
46.
47.     return createGraph(finalEdges, numOfNodes, numOfEdges,
48.     heuristics);
49. }
```

Изворни код 1

```

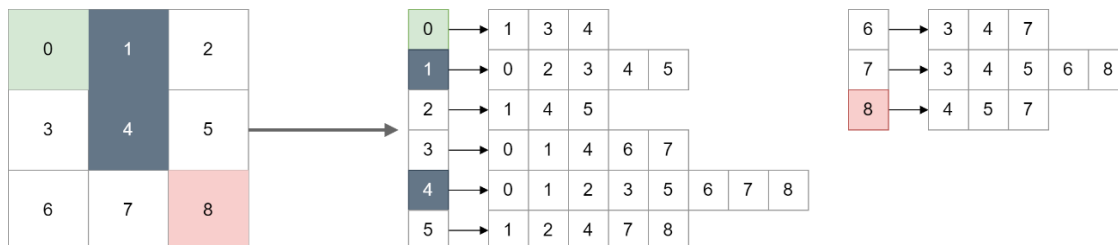
1. void addNeighbourCells(struct Cell** cells, int cellIndex,
   struct FullEdge** edges, int* numOfEdges){
2.     int i = cells[cellIndex]->i;
3.     int j = cells[cellIndex]->j;
4.
5.     float moves[8][3] = {{-1, -1, sqrt(2)}, {-1, 0, 1}, {-1, 1,
   sqrt(2)}, {0, -1, 1}, {0, 1, 1}, {1, -1, sqrt(2)}, {1, 0, 1},
   {1, 1, sqrt(2)}};
6.
7.     if(cells[cellIndex]->special)
8.         return;
9.
10.    for(int k = 0; k < 8; k++){
11.        int new_i = i + moves[k][0];
12.        int new_j = j + moves[k][1];
13.
14.        if(checkIfCellInsideBounaries(new_i, new_j) &&
   !cells[new_i * COLS + new_j]->special){
15.
16.            struct FullEdge* edge = (struct FullEdge*)
   malloc(sizeof(struct FullEdge));
17.            edge->src = cellIndex;
18.            edge->dest = new_i * COLS + new_j;
19.            edge->cost = moves[k][2];
20.
21.            edges[*numOfEdges] = edge;
22.            (*numOfEdges)++;
23.        }
24.    }
25. }

```

Изворни код 2

У фајлу *graph.c*, дата је функција *createGraph* која од сваког поља претходно описане матрице креира чвор графа, а од сваког комшије сваког поља креира ивицу графа. На слици 1 дата је визуелна представа транзиције из матрице поља у граф структуру података.

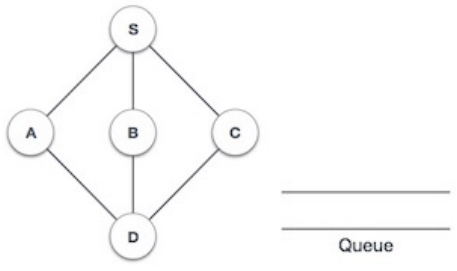
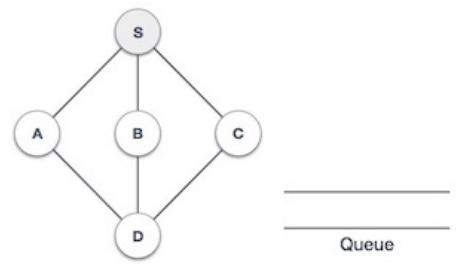
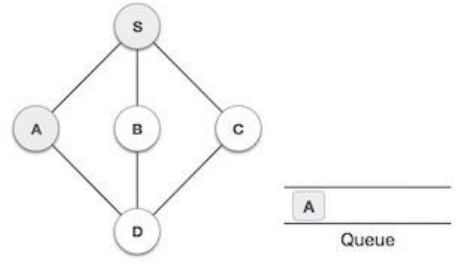
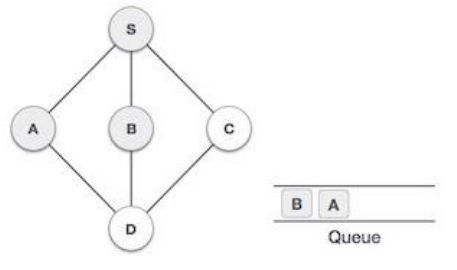
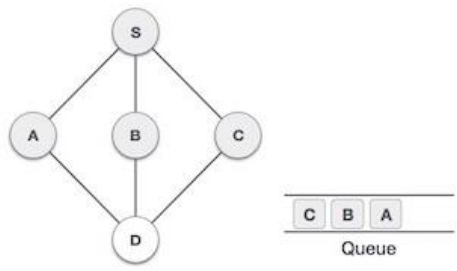
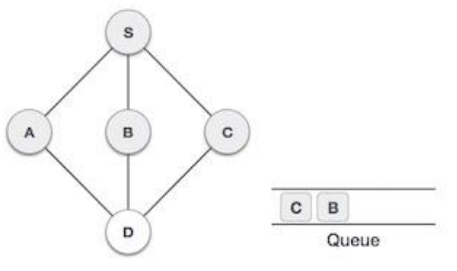
Након што је на овај начин креирана граф структура података, омогућена је визуелна презентација рада алгоритама претрага пријемчива за човеково око. Такође, промена величине графа је знатно олакшана, а то је врло битно за адекватно мерење перформанси између секвенцијалног и паралелног приступа претрагама.

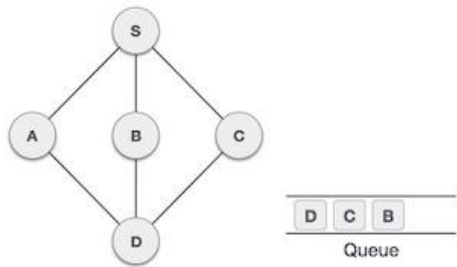
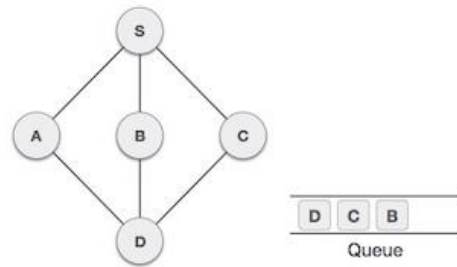


Слика 1. Трансформација матрице поља у граф структуру података

3. ПРЕТРАГА У ШИРИНУ

Претрага у ширину (енгл. *Breadth-first search*) је алгоритам за претрагу граф структура података и један је од важнијих алгоритама претрага с обзиром да представља основу за неке друге алгоритме [3]. Овај алгоритам почиње од коренског чвора графа и врши претрагу свих чворова на некој дубини k пре него што настави претрагу чворова на дубини $k + 1$. Као помоћна структура података, користи се ФИФО ред (енгл. *queue*). ФИФО ред подразумева да оно што се прво стави у њега, биће прво и добијено из њега. У табели 1 дата је визуелна презентација рада претраге у ширину над једним графом.

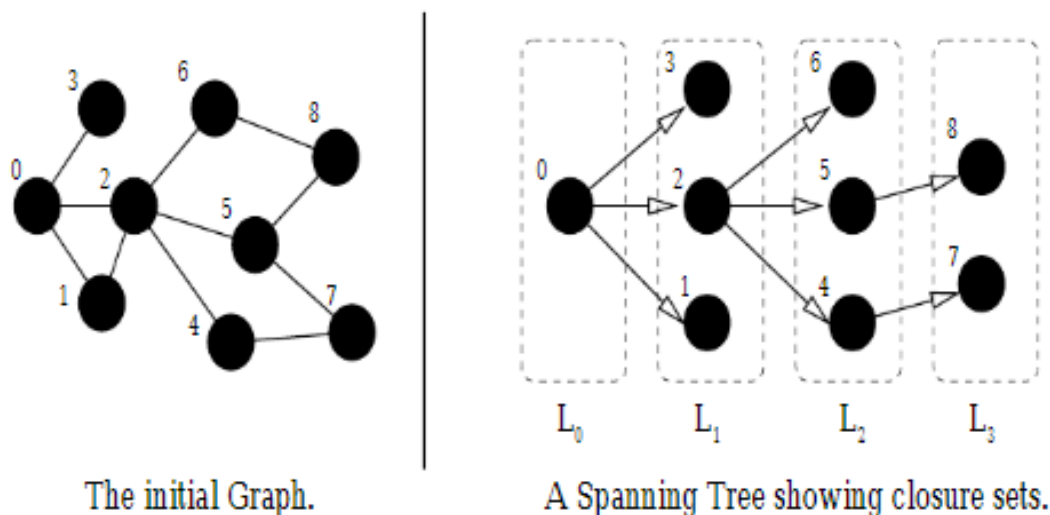
 <p>1. корак – Врши се иницијализација ФИФО реда</p>	 <p>2. корак – Почиње се од коренског чвора „S“ и он се обележава да је посећен</p>
 <p>3. корак – Чвор „S“ је повезан са чворовима „A“, „B“ и „C“. Као први чвор који се ставља у ФИФО ред и обележава као посећен, узима се чвор „A“</p>	 <p>4. корак – Следећи чвор који се посећује и ставља у ФИФО ред је чвор „B“</p>
 <p>5. корак – У ФИФО ред се додаје чвор „C“ као последњи чвор са којим је повезан коренски чвор „S“</p>	 <p>6. корак – Како више нема чворова са којим је повезан коренски чвор тј. обрађен је нулти ниво, из ФИФО реда</p>

	се ради извлачење првог чвора ит следећег нивоа дубине, а који ће због ФИФО принципа стављања и узимања из реда бити чвор „А“
 <p>7. корак – Чвор „А“ је повезан само само чвором „Д“, тако да се само он ставља у ФИФО ред за накнадну обраду</p>	 <p>8. корак – Иако више нема непосећених чворова, алгоритам се наставља примењивати све док се не испразни ФИФО ред</p>

Табела 1. Приказ рада претраге у ширину у корацима
(Слике преузете из [2])

На слици 2 је приказано на који начин претрага у ширину претражује по нивоима дубине. Ако је фактор гранања b , а дубина на којој се налази последњи чвор d , временска комплексност се рачуна помоћу израза (1) и износи $O(b^d)$, а максималан број чворова у меморији је једнак броју чворова на последњем нивоу d и то b^d .

$$1 + b^2 + b^3 + b^4 + \dots + b^d = O(b^d) \quad (1)$$



Слика 2. Претрага у ширину по нивоима дубине
(Слика преузета из [3])

3.1 Имплементација паралелне претраге у ширину

За имплементацију овог алгоритма, коришћен је C програмски језик којем су придодате компајлерске директиве које потичу од *OpenMP* АПИ-ја који подржава паралелизацију у дељеној меморији. Као помоћна структура података, коришћен је ФИФО ред који је имплементиран у фајлу *queue.c*. У овом фајлу се налазе функције *push_back* и *pop_front* које стављају и извлаче инстанце структуре *State* у ред, а то све по ФИФО принципу. Структура *State* ће бити објашњена у наставку. Такође, постоји и функција *isEmpty* која провера да ли је ред празан што ће бити од важности за заустављање претраге. Главнина претраге у ширину се налази у фајлу *bfs.c*, тачније у функцији *bfs*:

```
Struct Queue* bfs(
    struct Graph* graph,
    struct Queue* visitedQueue,
    int startNodeIndex,
    int endNodeIndex
)
```

Ова функција као параметре прима показивач на граф структуру података изгенерисану на основу матрице ћелија, показивач на ред у којем се чувају подаци о свим већобихћеним пољима што се користи приликом визуелне презентације путање, као и идентификатори почетног и крајњег чвора.

На самом почетку функције врши се добављање показивача на почетни и крајњи чвор на основу њихових индикатора прослеђених кроз параметре функције (*Изворни код 3*).

```
1. int numOfNodes = graph->numOfNodes;
2. struct Node* startNode = graph->head[startNodeIndex];
3. struct Node* endNode = graph->head[endNodeIndex];
```

Изворни код 3

У наставку се врши иницијализација *visited* низа типа *bool* у којем се чувају информације о томе да ли је неки чвор већ посећен или није. За ову иницијализацију искоришћена је *#pragma omp parallel for* директива заједно са клаузом *firstprivate(numOfNodes)* која преузима већ иницијализовану вредност за укупан број чворова (*Изворни код 4*).

```
1. bool visited[numOfNodes];
2.
3. #pragma omp parallel for firstprivate(numOfNodes)
4. for(int i = 0; i < numOfNodes; i++){
5.     visited[i] = false;
6. }
```

Изворни код 4

Потом долази до иницијализације почетног стања које се ставља на почетак тек иницијализованог ФИФО реда, а чвор на основу кога је настало иницијално стање се обележава као посећен (*Изворни код 6*). Структура *State* (*Изворни код 5*) којом се апстрахује стање је направљена са идејом да се помоћу ње има приступ свим следећим чворовима који се могу обићи, као и да се има приступ родитељским чворовима на основу којих се дошло до чвора који је апстрахован овим стањем што је од значаја приликом реконструкције целе путање доласка до крајњег чвора једном када се до њега дође, а о којој ће бити речи у наставку. Ова структура поседује и поље *cost* које није значајно за претрагу у ширину, али је врло значајно за A* алгоритам у којем је структура *State* такође коришћена.

```
1. struct State{
2.     struct Node* node;
3.     struct State* parent;
4.     float cost;
5. }
```

Изворни код 5

```
1. struct State* initialState = (struct State*) malloc(
2.     sizeof(struct State)
3. );
4. initialState->node = startNode;
5. initialState->cost = 0;
6. initialState->parent = NULL;
7.
8. struct Queue* queue = createQueue();
9.
10. push_back(queue, initialState);
11. visited[startNodeIndex] = true;
```

Изворни код 6

У „Изворном коду 7“ дат је остатак *bfs* функције. На самом почетку овог изворног кода дефинисан је ред који ће послужити за чување целокупне путање до крајњег чвора онда када он буде пронађен, као и променљива *found* која говори о томе да ли је путања пронађена или није и представља својеврсни индикатор завршетка претраге. Главни део овог изворног кода јесте *while* петља која обезбеђује да се код у њој извршава све док се ФИФО ред у потпуности не исразни или се не пронађе крајњи чвор. У наставку *while* петље инсталира се ред под називом *nextLevel* у који ће бити додати чворови из следећег нивоа дубине, а који представљају улаз за обраду у следећој

итерације *while* петље. Потом у 7-ој линији кода покреће се *for* петља која је уједно аотирана са *OpenMP* директивом *#pragma omp parallel for* која ће омогућити паралелно извршавање кода у самој петљи. Покретање *for* петље уместо *while* петље је омогућено знањем о укупној количини чланова реда о чему се води рачина приликом додавања и уклањања чланова из реда.

У наставку се врши добављање првог стања које је додато у ред и то стање се ставља у *visitedQueue* ред помоћу којег се ради визуализација обиђених поља. Добављање тог првог стања је неопходно аотирати са *#pragma omp critical* аотацијом како би се избегло вишеструко добављање и обрада идентичног чвора. Након што је добављено прво стање, проверава се да ли се дошло до крајњег чвора и ако јесте врши се реконструкција целе путање (Изворни код 9). Ако се није дошло до крајњег чвора, пролази се кроз све ивице тренутног чвора и уколико чворови који те ивице повезује нису већ посећени, у *nextLevel* ред се додаје ново стање са повезаним чвором и тренутним стањем као родитељским стањем. Након што се заврши обрада за све чворове са тренутне дубине, у *nextLevel* реду ће се налазити сва стања за наредну итерацију обраде следећег нивоа граф структуре података и овај ред се користи у следећој итерацији *while* петље. Приликом додавања нових стања у овај ред, потребно је водити рачуна да се не деси штетно преплитање, тако да је само додавање аотирано са *#pragma omp critical* аотацијом. У изворном коду 8 приказан је сам крај *bfs* функције који враћа путању ако постоји или враћа *NULL* вредност ако она не постоји.

```
1. struct Queue* path;
2. bool found = false;
3. while(!isEmpty(currentLevel) && !found){
4.
5.     struct Queue* nextLevel = createQueue();
6.
7.     #pragma omp parallel for
8.     for(int i = 0; i < currentLevel->size; i++){
9.
10.        struct State* currentState;
11.        #pragma omp critical
12.        currentState = pop_front(currentLevel);
13.        push_back(visitedQueue, currentState);
14.
15.        if(currentState->node->value == endNodeIndex){
16.            path = reconstructPath(currentState);
17.            found = true;
```

```
18.     }
19.
20.     struct Edge* edge = currentState->node->nextEdge;
21.
22.     while (edge != NULL){
23.         if(!visited[edge->dest]){
24.             struct State* nextState = (struct State*) malloc (
25.                 sizeof(struct State)
26.             );
27.             nextState->node = graph->head[edge->dest];
28.             nextState->cost = 0;
29.             nextState->parent = currentState;
30.
31.             #pragma omp critical
32.             push_back(nextLevel, nextState);
33.
34.             visited[edge->dest] = true;
35.         }
36.
37.         edge = edge->next;
38.     }
39. }
40.
41. free(currentLevel);
42. currentLevel = nextLevel;
43. }
```

Изворни код 7

```
1. if(cancel)
2.     return path;
3. else
4.     return NULL;
```

Изворни код 8

Помоћу ове *bfs* функције континуирано се пролази кроз све ивице чворова које се налазе на неком нивоу k , оне се додаје на крај ФИФО реда и бивају обрађене након што се обраде сви чворови нивоа k , и као такве, чиниће ниво $k + 1$. Овакво обрађивање по нивоима је идеја која стоји иза претраге у ширину и оно се врши све док се не дође до крајњег жељеног чвора, након чега је неопходно реконструисати целу путању на основу које се и дошло до крајњег чвора. Та реконструкција путање се своди на кретање кроз све родитељске чворове све док се не дође до стања које нема свог родитеља, а то је иницијално стање од којег се i кренуло (*Изворни код 9*).

```
1. struct Queue* reconstructPath(struct State* bestState){
2.     struct Queue* path = createQueue();
3.     while(bestState->parent != NULL) {
4.         push_back(path, bestState);
5.         bestState = bestState->parent;
6.     }
7.     push_back(path, bestState);
8.
9.     return path;
10. }
```

Изворни код 9

3.2 Поређење перформанси

У табели 2, дата су поређења брзина извршавања програма између секвенцијалног и паралелног приступа. У заглављима колона су дате величине матрице ћелија. За свако извршавање програма, генерисане препреке су биле идентичне, а задатак је увек био да се стигне од почетне, нулте ћелије до ћелије која се налази у последњем реду и колони. Оно што је приметно је да паралелизација алгоритма није довела ни до какве предности у брзини извршавања, већ је чак довела до успорења претраге. Једини случај када је паралелни приступ дао бољи резултат јесте случај када је потребно извршити неку захтевну обраду над стањем које је извучено из реда. Као симулације овакве обраде и оптерећења коришћена је једноставна *for* петља која је имала за циљ да велики број пута дода вредност 1 на једну променљиву типа *int*. Паралелни приступ је овде показао значајно убрзање из разлога што се оваква обрада може вршити независно једна од друге за сваки чвор те је има смисла извршавати на више процесорских језгара, док се код секвенцијалног приступа врши за један по један чвор на само једном језгру.

Паралелизација претраге у ширину и A* алгоритма

	100 × 100	500 × 500	1000 × 1000	20 × 20 са обрадом
Секвенцијални приступ	0,01с	0,58с	4,87с	32,14с
Паралелни приступ	0,02с	2,06с	7,64с	7,72с

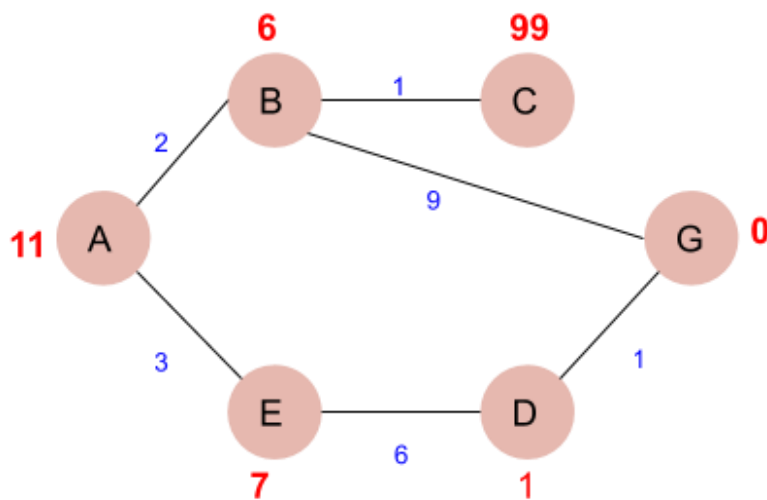
Табела 2. Поређење перформанси између секвенцијалног и паралелног приступа

4. A* АЛГОРИТАМ

A* алгоритам је још један алгоритам за обилазак граф структуре података и проналажење путање од почетног до крајњег чвора. За разлику од претраге у ширину, A* алгоритам уводи појам хеуристике која помаже у одређивању најбоље путање до циља. Што је хеуристика квалитетнија, то је и квалитет путање бољи. Након што се крене од коренског чвора, за сваку ивицу се рачуна вредност f која представља збир вредност g и h , где g представља суму цена неопходних да се дође до посматраног чвора, а h вредност која се често зове хеуристиком, представља процену цене да се дође од посматраног чвора до крајњег чвора. Као процена цене, она не говори о тачној удаљености с обзиром да се на путу до циља могу наћи разноразне препреке, али представља додатно знање о проблему које може побољшати процес претраге. Као хеуристика, у практичном делу овог рада, коришћена је еуклидова дистанца која се рачуна према изразу (2). Оно што је битно напоменути јесте да хеуристика не би требала да прецењује удаљеност од циља јер то може довести до ситуације да алгоритам ради на обилажења чворова који заправо неће бити део оптималне руте.

$$h = \sqrt{(currentCell.x - endCell.x)^2 + (currentCell.y - endCell.y)^2} \quad (2)$$

Од свих чворова који се налазе у реду за обрађивање, бира се онај чвор који има најмању f вредност. Потом се за његове повезане чворове рачунају f вредности које се стављају у ред, а које ће утицати на одлуку који чвор би требало следећи обрадити. Обилажење чворова се врши све док се не дође до ситуације да је чвор са најмањом f вредношћу заправо крајњи, жељени чвор. На слици 3 је дата једна граф структура података на којој ће бити објашњен рад A* алгоритма.



Слика 3. Приказ граф структуре података на којој је демонстриран рад A* алгоритма
(Слика преузета са [4])

Свим чворовима су придодате вредности обележене црвене бојом, а које представљају вредности хеуристике, а чворови су међусобно повезани ивицама које су квантификоване плавим бројевима и представљају цену да се

из једног чвора стигне у други. Креће се од чвора „А“, а потребно је стићи до чвора „Г“. Како је чвор „А“ иницијални чвор, цена да се стигне до њега, тј. g вредност је једнака 0, тако да ће f вредност овог чвора бити једнака само његовој хеуристичкој вредности, а то је 11. Од чвора „А“ може се стићи до чвора „В“ по цени 2 што ће са 6 као хеуристичком вредношћу чвора „В“ дати укупно 8 за f вредност путање $A \rightarrow B$. Чвор „Е“ је такође повезан са чвором „А“ и та путања ће имати f вредност 9. Како је f вредност путање $A \rightarrow B$ мања од f вредности путање $A \rightarrow E$, као следећи чвор који се обрађује узима се чвор „В“, а f вредност путање $A \rightarrow E$ се и даље чува у случају да она у једном тренутку постане најмања. Од чвора „В“ се може стићи до чворова „С“ и „Г“ чије путање ће као f вредности имати вредности $(2 + 1) + 99 = 102$ и $(2 + 9) + 0 = 11$, респективно. Битно је приметити да у калкулацију f вредности улази и цена да се стигне до чвора „В“, а потом у наредне чворове, а не само појединачне цене на ивицама. У овом тренутку се стиже до жељеног чвора „Е“, међутим, претраге се не завршава јер постоји путања која има мању укупну f вредност ($A \rightarrow E$), а која може довести до јефтиније путање. Сада се посматра путања $A \rightarrow E$. Од чвора „Е“ се може стићи једино до чвора „Д“ који ће дати f вредност 10, што је мање од f вредности путање $A \rightarrow B \rightarrow G$, тако да ће се чвор „Д“ узети у разматрање. Од чвора „Д“ се може стићи до жељеног чвора „Г“, а целокупна путања ће имати f вредност $(3 + 6 + 1) + 0 = 10$ што је уједно и најјефтинија путања која ће бити проглашена за оптималну и крајњу.

4.1 Имплементација паралелног A* алгоритма

Као помоћна структура, коришћен је приоритетни ред (енгл. *priority queue*) уместо ФИФО реда коришћеног приликом имплементације претраге у ширину. Приоритет се односи на то да се оне путање које имају мању f стављају на почетак реда, тако да се извлачење из њега врши у $O(1)$ временској комплексности. Оно што је мана тренутног приступа је то што временска комплексност за свако додавање у ред $O(n)$, што је лошије од структура података попут хипа (енгл. *heap*) који за додавање има временску комплексност $O(\log N)$. Даља оптимизација тренутног решења би свакако водила ка превођењу тренутног реда у хип структуру података. Сама имплементација A* алгоритма је поприлично слична имплементацији претраге у ширину, а разлика се најпре односи на рачуње цене да се дође до неког чвора. Код претраге у ширину, цена је увек била 0, док је код A* алгоритма неопходно водити рачуна о суми свих цена до тренутног чвора где помаже већ поменута структура *State* у којој се чувају акумулиране вредности цена до неког посматраног чвора. Главна функција у *astar.c* фајлу у којем је дефинисан рад A* алгоритма јесте функција *astar* која прима идентичне параметре као и претходно поменута *bfs* функција:

```
struct Queue* aStar(
    struct Graph* graph,
    struct Queue* visitedQueue,
    int startNodeIndex,
    int endNodeIndex
)
```

Сам почетак ове функције (*Изворни код 10*) је поприлично сличан почетку *bfs* функције и своди се на иницијализацију почетног и крајњег чвора, иницијализацију иницијалног стања, као и иницијализацију низа који ће водити рачуна о већ посећеним чворовима што се може одрадити на паралелан начин коришћењем *#pragma omp parallel for* анотације из *OpenMP* АПИ-ја.

```

1. struct Queue *queue = createQueue();
2.
3. struct Node* startNode = graph->head[startNodeValue];
4. struct Node* endNode = graph->head[endNodeValue];
5.
6. struct State* initialState = (struct State*) malloc(
7.     sizeof(struct State)
8. );
9. initialState->node = startNode;
10. initialState->cost = 0;
11. initialState->parent = NULL;
12.
13. bool visited[graph->numOfNodes];
14. #pragma omp parallel for
15. for(int i = 0; i < graph->numOfNodes; i++){
16.     visited[i] = false;
17. }
18.
19. push(queue, initialState);

```

Изворни код 10

Наставак функције (*Изворни код 11*) се своди на *while* петљу која извршава код унутар ње све док се не испразни приоритетни ред или се не дође до жељеног чвора. Унутар петље се скида прво стање из приоритетног реда (*Линија 5*) које ће уједно бити и најбоље по *f* вредности због начина на који се врши додавање у ред где се води брига о томе да стања са најмањим *f* вредностима буду на почетку. Провера се да ли је нови чвор до којег се дошло крајњи чвор (*Линија 10*). Ако јесте, потребно је урадити реконструкције целе путање до њега на идентичан начин како је то рађено код претраге у ширину. У линијама 15-18, чвор се обележава да је посећен или се прелази на наредни чвор ако је већ посећен. На самом крају се улази у паралелни регион који би требало да направи нова стања за све ивице чвора који је тренутно на обради

узимајући у обзир цену да се пређе у следећи посматрани чвор заједно са целокупном ценом да се дође до њега. Ово је имплементирано на тај начин да једна нит креира задатке који ће извршавати претходно описани посао, а потом нити узимају задатке из реда задатака и извршавају их.

```
1. bool found = false;
2. struct Queue* path;
3.
4. while(!isEmpty(queue) && !found){
5.     struct State* bestState = pop(queue);
6.     push(visitedQueue, bestState);
7.
8.     struct Node* currentNode = bestState->node;
9.
10.    if(currentNode == endNode){
11.        path = reconstructPath(bestState);
12.        found = true;
13.    }
14.
15.    if(visited[currentNode->value])
16.        continue;
17.    else
18.        visited[currentNode->value] = true;
19.
20.    struct Edge* edge = currentNode->nextEdge;
21.    #pragma omp parallel
22.    {
23.        #pragma omp single
24.        {
25.            while(edge != NULL){
26.                #pragma omp task
27.                {
28.                    struct State* possibleState = (struct
State*) malloc(sizeof(struct State));
```

```

29.           possibleState->node = graph->head[edge-
>dest];
30.           possibleState->cost = bestState->cost +
edge->cost;
31.           possibleState->parent = bestState;
32.
33.           #pragma omp critical
34.           {
35.               push(queue, possibleState);
36.           }
37.       }
38.
39.       edge = edge->next;
40.   }
41. }
42. }
43. }
44.
45. if(!found) return NULL;
46. else return path;

```

Изворни код 11

4.2 Поређење перформанси

У табели 3, дате су брзине извршавања програма код секвенцијалног, као и паралелног приступа за различите величине почетне матрице ћелије и идентичне препреке. Оно што је приметно јесте да паралелни приступ није донео побољшања. Не само да није донео побољшања, већ је претрага доста успоренија у односу на секвенцијални приступ. Интуиција која лежи иза ове појаве је та да је A* алгоритам доста секвенцијалан по својој природи јер обрада сваког наредног корака зависи од претходног корака, те не постоји много простора за изоловање послова које нити могу независно да раде.

	100 × 100	500 × 500	1000 × 1000
Секвенцијални приступ	0,21с	7,87с	38,81с
Парелелни приступ	0,32с	10,46с	50,05с

Табела 3. Поређење перформанси између секвенцијалног и паралелног приступа

5. ЗАКЉУЧАК

У овом раду приказан је један пруступ паралелизацији претраге у ширину и A* алгоритма. Као делови кода који су узети у обзир за паралелизацију јесу додавање нових чворова тј. стања у ФИФО и приоритетне редове. Разлог зашто се баш ови делови алгоритама паралелизовани лежи у томе што се они могу посматрати независно једни од других тако да постоји простор за паралелизацију. Упркос томе, само довање чворова у редове не захтева никакву комплексну или захтевну обраду, тако да су паралелана решења у свим случајевима дала спорије резултате него секвенцијална. Једини случај када је паралелизација помогла у убрзању рада претраге јесте случај када је симулирана захтевнија обрада чворова који су добијени из ФИФО реда код претраге у ширину из разлога што се оваква обрада могла вршити паралелно на више процесорских језгра уместо секвенцијално на једном језгру.

ЛИТЕРАТУРА

- [1] <https://towardsdatascience.com/search-algorithm-breadth-first-search-with-python-50571a9bb85e>, Chao De-Yu – [Пристапано јануара 2022. године]
- [2] https://www.tutorialspoint.com/data_structures_algorithms/breadth_first_traversal.htm - [Пристапано јануара 2022. године]
- [3] Jason J Holdsworth, „*The Nature of Breadth-First Search*“
- [4] <https://www.mygreatlearning.com/blog/a-search-algorithm-in-artificial-intelligence/> - [Пристапано јануара 2022. године]