TÉCNICO LISBOA

Parallel and Distributed Computing

Project Assignment

TRAVELING SALESPERSON

.

BRANCH & BOUND

Version 1.1 (20/02/2023)

2022/2023

3rd Quarter

Contents

Revisions

Version 1.1	Added section on internal requisites (Section 3.3) New disambiguation rules implied minor revision of the example Fixed pseudocode, removed $+1$ in line 10
Version 1.0 (February 13, 2023)	Initial version

1 Introduction

The purpose of this class project is to give students a hands-on experience in parallel programming on both shared-memory and distributed-memory systems, using OpenMP and MPI, respectively. For this assignment you are to write a sequential and two parallel implementations of a program that computes the shortest route to visit a given set of cities, the well-known traveling salesperson problem.

As you may recall, this optimization problem falls in the NP-Hard complexity class, meaning that all known solutions to solve it exactly imply an exhaustive search over all possible sequences. In this project we will be using an estimated lower-bound that allows pruning the search whenever a partial solution has a lower bound higher than the cost of a previously found solution.

2 Branch and Bound Algorithm

The algorithm to be implemented consists in building a search tree which covers all possible sequences of cities. However, using estimated lower bounds, branches of this tree will be discarded when it is guaranteed that no solution better than the current one can be found down that branch of the tree.

Moreover, we will use a priority queue so that the most promising branches are examined first, giving priority to nodes with lower lower-bound.

2.1 Search Tree

Algorithm 1 presents the pseudocode for building the search tree. It receives three input parameters:

Distances: input matrix where entry (i, j) corresponds to the distance (or cost) between cities i and j;

N: number of cities;

BestTourCost: maximum cost allowed for solution, *i.e.*, not interested in solutions worse than this value.

The algorithm is based on a priority queue. Without loss of generality, we will be looking for paths that start at city 0. Hence, we initialize the queue with an entry with the following parameters:

- path initialized with a single entry, city 0;
- cost of current path, 0 at this point as no edges have been traversed;
- an initial lower bound for the input matrix (see Section 2.2);
- number of nodes in current path, in this case 1;
- current city, which is 0.

We then enter a loop where at each iteration we remove the first entry of the queue and analyze it. Since this is a priority queue, we will remove the most promising of the already visited nodes of the search tree, that is, the one with lowest lower-bound.

The first thing to check is if the lower-bound of this entry is higher than the best solution that has been found so far. If this is the case, then we are done, as all other entries in the queue have higher lower-bounds than the current one and thus cannot be better than the solution we already have.

Algorithm 1 TSP Branch and Bound

```
1:  procedure TSPBB(Distances, N, BestTourCost)
2:      Tour ← {0}                                          ▷ Start at city 0
3:      LB ← lower bound on TSP tour cost for graph Distances
4:      Queue ← (Tour, 0, LB, 1, 0)          ▷ Tour, Cost, Bound, Length, Current city
5:      while Queue ≠ {} do
6:          (Tour, Cost, Bound, Length, Node) ← Queue.pop()
7:          if Bound ≥ BestTourCost then          ▷ All remaining nodes worse than best
8:              return BestTour, BestTourCost
9:          end if
10:         if Length = N then                    ▷ Tour complete, check if it is best
11:             if Cost + Distances(Node, 0) < BestTourCost then
12:                 BestTour ← Tour ∪ {0}
13:                 BestTourCost ← Cost + Distances(Node, 0)
14:             end if
15:         else
16:             for each neighbor v of Node and v ∉ Tour do
17:                 newBound ← updated lower bound on tour cost
18:                 if newBound > BestTourCost then   ▷ Cannot be better than best so far
19:                     Continue
20:                 end if
21:                 newTour ← Tour ∪ {v}
22:                 newCost ← cost + Distances(Node, v)
23:                 Queue.add((newTour, newCost, newBound, Length + 1, v))   ▷ By bound
24:             end for
25:         end if
26:     end while
27:     return BestTour, BestTourCost
28: end procedure
```

Next we check if this entry completes a tour. If so, we compare this tour with the previous best solution found and save it if it has lower cost. Otherwise we discard it.

If none of the previous conditions apply, we visit all the cities that we can get to from the current one and that are not yet in the current path. For each we compute a new lower-bound (again, see Section 2.2). If this lower-bound is above the best solution, no better solution exists down this path so we do not pursue it. Otherwise, we create a new path and insert, in order, in the queue.

2.2 Computing the Lower Bounds

The initial overall lower-bound can be determined using the following observation: for the tour we need two edges for each city (one to enter, another to leave) and no solution can be better than using the two edges with lowest cost for each city. If $min1(i)$ and $min2(i)$ represent the adjacent edges with lowest cost and second lowest cost of city i, the initial lower bound, LB, is simply computed using:

$$LB = \frac{1}{2} \sum_i min1(i) + min2(i)$$

(the $\frac{1}{2}$ is because we are counting each edge twice, the edge leaving one city is entering another).

Then, every time a new city is visited we need to update this bound, accounting for the actual cost of the edge, and removing the estimate we considered initially. To keep the lower-bound as tight as possible, we remove the highest of the minimum edges considered for the nodes that is below the cost of the edge we are traversing. If we are visiting city t from city f, this translates to the following operation:

$$cf = \begin{cases} min2(f) & \text{if } cost(f,t) \geq min2(f) \\ min1(f) & \text{otherwise} \end{cases} \qquad ct = \begin{cases} min2(t) & \text{if } cost(f,t) \geq min2(t) \\ min1(t) & \text{otherwise} \end{cases}$$

$$newLB = LB + cost(f,t) - (cf + ct)/2$$

2.3 Priority Queue

We will provide the code that implements the priority queue, using a binary heap, that provides the following operations in C and C++:

C	C++	Complexity
`queue_create()`	`PriorityQueue<T>`	$O(1)$
`queue_delete()`	NA	$O(1)$
`queue_push()`	`Queue.push()`	$O(\log n)$
`queue_pop()`	`Queue.pop()`	$O(\log n)$
`queue_print()`	`Queue.print()`	$O(n \log n)$

where n is the number of elements in the queue.

3 Implementation Details

3.1 Input Data

Your program should take two command line parameters:

```
$ tsp <cities file> <max-value>
```

The `<cities file>` has the information about the distances (cost) between cities. The first line of the file has two positive integers, indicating the number of cities and total number of roads. Then each line represents one road between two cities and has three integers, the first two are the indexes of the cities and the third the distance between them. Note that we assume all roads are bidirectional, they connect the cities both ways.

The `<max-value>` specifies a maximum value that we accept for the solution.

3.2 Output Data

Your program should send to the standard output, `stdout`, the cost of the best tour and the sequence of cities to visit in this tour. The first value should be by itself in the first line, using one decimal digit. The tour should come in the line below, a sequence of integers representing the cities separated by a single space, starting and ending at 0.

If no solution is found, either because the graph is disconnect or no solution better than the specified `<max-value>` exists, the program should output `NO SOLUTION`.

The submitted programs should only print this information (and **nothing else!**) to the standard output, so that we can validate the result automatically.

The project **cannot be graded** unless you follow strictly these input and output rules!

3.3 Internal Standardization Requisites

Note that the distances (cost) are real numbers. Please make sure to use the type `double` for all your computations.

In case two nodes of the tree have the same lower-bound value, use the index of the city to break the tie, lower indices should be given higher priority. (it is still possible to have a draw, but very unlikely, hence you may safely ignore that situation)

To ensure you still use a priority queue, it is disallowed to use the priority feature of tasks in OpenMP.

3.4 Measuring Execution Time

To make sure everyone uses the same measure for the execution time, the routine `omp_get_wtime()` from OpenMP will be used, which measures real time (also known as "wall-clock time"). This same routine should be used by all three versions of your project.

Hence, your programs should have a structure similar to this:

```
#include <omp.h>
<...>

int main(int argc, char *argv[])
{
    double exec_time;

    parse_inputs();
    exec_time = -omp_get_wtime();

    tsp();

    exec_time += omp_get_wtime();
    fprintf(stderr, "%.1fs\n", exec_time);

    print_result();       // to the stdout!
 }
```

In this way the execution time will only account the algorithm running time, and be sent to the standard error, `stderr`. This separation allows, if needed, sending the standard output to `/dev/null`.

Because this time routine is part of OpenMP, you need the include `omp.h` and compile all your programs with the flag `-fopenmp`.

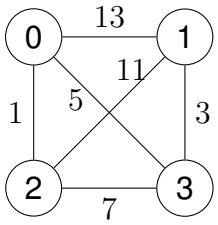

Figure 1: Interconnect graph for example `ex2.in`.

3.5 Sample Problem

Assume you have a file `ex2.in` with the following content:

```
4 6
0 1 13
0 2 1
0 3 5
1 2 11
1 3 3
2 3 7
```

The interconnect graph corresponding to this input file is presented in Figure 1.

To find the best tour in this graph, and using a starting lower-bound of 40, your program should be executed using the following command, and obtain this output:

```
$ tsp ex2.in 40
20.0
0 3 1 2 0
$
```

The search tree corresponding to this example is presented in Figure 2. The figure shows the lower bound, LB, for each node of the tree, and the cost of each tour at the leaves. The traversal of this tree is the following:

1. Node 0: all children of 0 are created and inserted into the queue. Because we use a priority queue, the first child of 0 to explore could be 2 or 3 because they both have the lowest lower-bound. We visit 2 first because it has a lower index;
2. Node 2 (level 2): its children, 1 and 3, are visited (0 is already in the path), their lower-bounds computed and added to the queue. Now, the first node 3 has the lowest lower-bound of all nodes in the queue (18) and is processed;
3. Node 3 (level 2): similarly, children 1 and 2 are visited, and 1 now has the lowest bound (18);
4. Node 1 (level 3): this node only has one child not in its current path, hence only node 2 is visited and added to the queue. We have two nodes in the queue with lower-bound 19, node 2 at level 3 is picked next (lower index);
5. Node 2 (level 3): again, there is only one node not in the path, so node 1 is added to the queue with a lower-bound of 21;

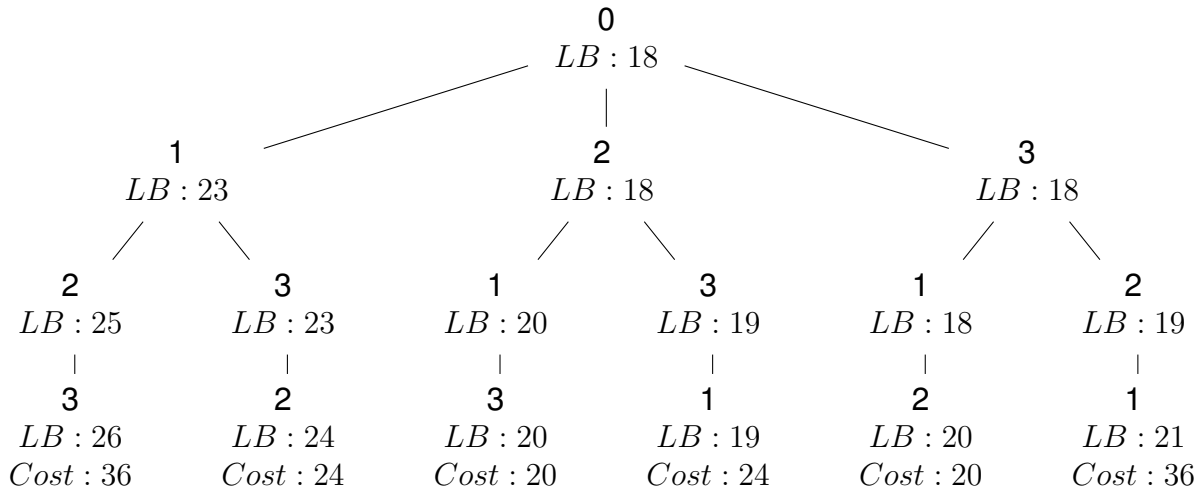

Figure 2: Search tree for example `ex2.in`.

6. Node 3 (level 3): node 1 is also the only not in the current path, it is visited and added to the queue, now the one with lowest lower-bound, 19;
7. Node 1 (level 4): tour completed, we save this tour $\{0, 2, 3, 1\}$ with a cost of 24. We now have in the queue two nodes with lower-bound 20, we pick node 1 at level 3;
8. Node 1 (level 3): we visit its only child 3, with the same lower-bound (20) of node 2 at level 4, we pick the latter;
9. Node 2 (level 4): again, we complete a tour with cost 20, better than the previous, so $\{0, 3, 1, 2\}$ becomes the current solution;
10. Node 3 (level 4): this node has lower-bound 20, equal to the solution we already have, and it is the lowest in the queue, meaning there are no better tours possible, terminating our search.

Note that if we run this instance with a maximum value of 10, the result should indicate that no such tour exists:

```
$ tsp ex2.in 10
NO SOLUTION
$
```

We will have available several public instances for you to compare your results against. Naturally, the evaluation for grading will be carried out on a set of private instances.

4 Part 1 - Serial implementation

Write a serial implementation of the algorithm in C or C++. Name the source file of this implementation `tsp.c`. As stated, your program should expect two input parameters.

Make sure to include a Makefile, the simple command

```
$ make
```

should generate an executable with the same name as the source file (minus the extension). Also, to be uniform across groups, in this makefile please use the `gcc` compiler with optimization flag `-O3`.

This version will serve as your base for comparisons and must be as efficient as possible.

5 Part 2 - OpenMP implementation

Write an OpenMP implementation of the algorithm, with the same rules and input/output descriptions. Name this source code `tsp-omp.c`. You can start by simply adding OpenMP directives, but you are free, and encouraged, to modify the code in order to make the parallelization as effective and as scalable as possible. Be careful about synchronization and load balancing!

Important note: in order to test for scalability, we will run this program assigning different values to the shell variable `OMP_NUM_THREADS`. Please do **not** override this variable or set the number of threads in your program, otherwise we will not be able to properly evaluate it!

6 Part 3 - MPI implementation

Write an MPI implementation of the algorithm as for OpenMP, and address the same issues. Name this source code `tsp-mpi.c`.

For MPI, you will need to modify your code substantially. Besides synchronization and load balancing, you will need to create independent tasks, taking into account the minimization of the impact of communication costs. You are encouraged to explore different approaches for the problem decomposition.

Extra credits will be given to groups that present a combined MPI+OpenMP implementation.

7 What to Turn in, and When

You must eventually submit the sequential and both parallel versions of your program (**please use the filenames indicated above**), and a table with the times to run the parallel versions on input data that will be made available (for 1, 2, 4 and 8 parallel tasks for both OpenMP and MPI, and additionally 16, 32 and 64 for MPI).

For both the OpenMP and MPI versions (not for serial), you must also submit a short report about the results (2-4 pages) that discusses:

- the approach used for parallelization
- what decomposition was used
- what were the synchronization concerns and why
- how was load balancing addressed
- what are the performance results, and are they what you expected

The code, makefile and report will be uploaded to the Fenix system in a zip file. **Name these files** as `g<n>serial.zip`, `g<n>omp.zip` and `g<n>mpi.zip`, where `<n>` is your group number.

1st due date, serial version: **March 10th**, until 23:59.

2nd due date (OpenMP): **March 24th**, until 23:59.

3rd due date (MPI): **April 14th**, until 23:59.