Arthur Wuterich, Cameron Hall
CS425, Homework 6/7
Fall 2015
Professor Riviore

- Algorithm Choice and Serial Implementation
  - Algorithm Specifics
    - The algorithm we chose to solve the Traveling Salesman Problem is a greedy branch-and-bound inclusion-exclusion algorithm.
    - Algorithm Example
      - For this example I will complete the algorithm using the following cost matrix for a solution with 5 cities:

```
Initial cost matrix for algorithm:
-  6  1  6  6
10 -  2  9  4
5  4  -  6  1
2  3  6  -  9
6  6  10 9  -
```

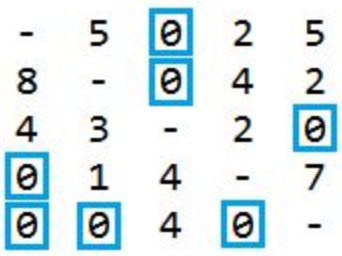
We will perform an alignment on the matrix which is the process of reducing each value of the matrix based on the row minimums and then the column minimums.

```
Cost matrix after first align
-  5  0  2  5
8  -  0  4  2
4  3  -  2  0
0  1  4  -  7
0  0  4  0  -
```

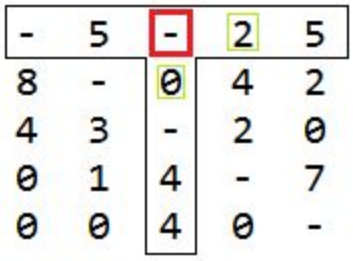
This will give us a rough lower bound:

```
With initial lower bound of 15
```

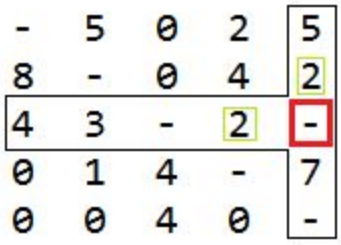
The algorithm starts operation with the cost matrix after the first align. We consider each entry of the cost matrix that has a 0 as these are the paths with the lowest cost and are suitable to be considered into the final solution.



For each potential candidate we compute the maximum change from the exclusion alternative and maximize this value. An example of using the first potential candidate(2,0):



Where the new row minimum is 2 and column minimum is 0 for a total difference of 2. We repeat this process for each candidate and return the candidate first experienced with the maximum difference value. This final candidate which maximises this difference is(2,4) with a value of 4:



Path 2:4 will be our selection for considering inclusion-exclusion.

Before we generate the inclusion and exclusion nodes we need to check the candidate path will not cause a cycle in the solution we have generated so far. Right now our solution is empty but the general principle is we will make sure that the destination city does not form a path that includes the source city. If this is the case then including the path will cause a cycle. If a cycle is detected then we modify the cost matrix by excluding the candidate path and recalculating the lower bound and candidate.

Now we generate the Inclusion node by generating a cost matrix where the path is selected and considered within the solution. After we update the matrix we need to assign the cost of reaching this node which is the lower bound of the original matrix with the new lower bound of the inclusion matrix (15 + 0):

```
Candidate path for algorithm: 2:4, with number of alternative paths: 0
Create right(inclusion) child with cost: 15, matrix:
-  5  0  2  -
8  -  0  4  -
-  -  -  -  -
0  1  4  -  -
0  0  -  0  -
```

This effectively reduces the problem from a NxN TSP problem to a (N-1)x(N-1) TSP problem.

Next we generate the exclusion node by removing the value at (2,4) which essentially removes the edge from consideration. The cost is calculated the same as the inclusion where an alignment is performed after removing the edge(15 + 4):

```
Create left(exclusion) child with cost: 19, matrix:
-  5  0  2  3
8  -  0  4  0
2  1  -  0  -
0  1  4  -  5
0  0  4  0  -
```

Now that we have both the inclusion and exclusion node we choose the path with the better lower bound which in our case is the inclusion path. We accept the path from 2 to 4 *(city 3 -> city 5)* into the solution and repeat the algorithm using the matrix generated for the inclusion branch.

```
Inclusion has a better lower bound and we will include this path into solution.
Adding the path: 2:4, into the solution
```

If the exclusion branch proves to have a better lower bound we will repeat the algorithm from the exclusion branch(where the considered path has been removed from consideration) until the final solution path is the size of the number of cities. Final solution:

```
Solution Path: City 1 -> City 4 -> City 2 -> City 3 -> City 5 -> City 1
Total path length: 5, with cost: 18
```

You can find the full listing of this example in the root of the project

- Algorithm Pseudocode
  - ```
    M = Initial Cost Matrix
    Step = Current Step of Algorithm
    G = Path Set
    Step.matrix = M
    Step.cost = align(Step.matrix)
    While len(G) < numberOfCities:
          candidate = findCandidate(Step.matrix)
          while candidate creates loop in G:
                exclude candidate from solution
                Step.cost += align(Step.matrix)
                candidate = findCandidate(Step.matrix)

          inclChild = copy(Step)
          include candidate in solution
          inclChild.cost += align(inclChild.matrix)
          exclChild = copy(Step)
          exclude candidate in solution
          exclChild.cost += align(exclChild.matrix)

          if inclChild.cost < exclChild.cost:
                G.include(candidate)
                Step = inclChild
          else:
                Step = exclChild
    ```
  - findCandidate is explained in the working example
  - including and excluding candidates is explained in the working example
- **Serial Code and Test Infrastructure**
  - *[Code and Driver located in serial folder on project root]*
  - To use the test infrastructure run **driver.py** which will run the serial version with two inputs of 50 and 250.
    ```

- Parallel Implementation
    - *[Code and Driver location in parallel folder on project root]*
    - **Parallelization Strategy**
        - Because of the algorithm choice our parallel implementation will optimize parts of the algorithm. Our selection of optimizations will be based on using the profiling utility **callgrind**(plugin for valgrind) for inputs of 300 and 500. This profile data is available from the profile directory at the project root.
        - The first iteration showed that the two functions **findMinInRow** and **findMinInCol** accounted for around 44% of the execution time. OpenMP will be used to parallelize the two functions where each thread will be expected to perform a local min of n/P elements with one critical section returning the global min.
        - The second iteration showed that the **findCandidate** function was the next largest user defined function taking 0.63% of the execution time. OpenMP will be used to parallelize this function where each thread will perform a row-wise findCandidate with one critical section to return the maximum exclusion value candidate.
    - To use run **driver.py <threads>** which will run the parallel version using the specified number of threads on two inputs of 50 and 250.
    - **Performance Measurements (see graphs below)**
        - Overall, performance increases were most significant when the input size was larger. Running the parallel portions of our algorithm across 4 threads with an input size of 250 gave the largest speedup of 1.795.
        - When run with 16 threads, our algorithm performed significantly worse with either input size. This suggests that our algorithm is not strongly scalable. If we could better distribute the cost matrix amongst threads in **findMinInRow** and **findMinInCol**, perhaps we could make better use of additional threads.
        - Our algorithm uses additional hardware resources more efficiently utilized when our problem size is increased, suggesting that our algorithm is weakly scalable.
        - When running profiling tools against our algorithm, we noticed that calls to STL container functions make up a non-trivial portion of our execution time. One potential strategy for further optimization is to use statically allocated c-style arrays instead of *std::vector* whenever possible. This would allow the cost matrix to be stored contiguously on the stack rather than on the heap and could lead to improvements in cache performance.
        - The nature of this algorithm makes it difficult to parallelize at a high level. Exploration of alternative parallelization strategies for **findMinInRow, findMinInCol, and findCandidate** could potentially lead to better performance.

- Investigation into additional portions of the algorithm that can be parallelized could also lead to performance increases.