Floyd-Warshall Algorithm

Ethan Soo Hon

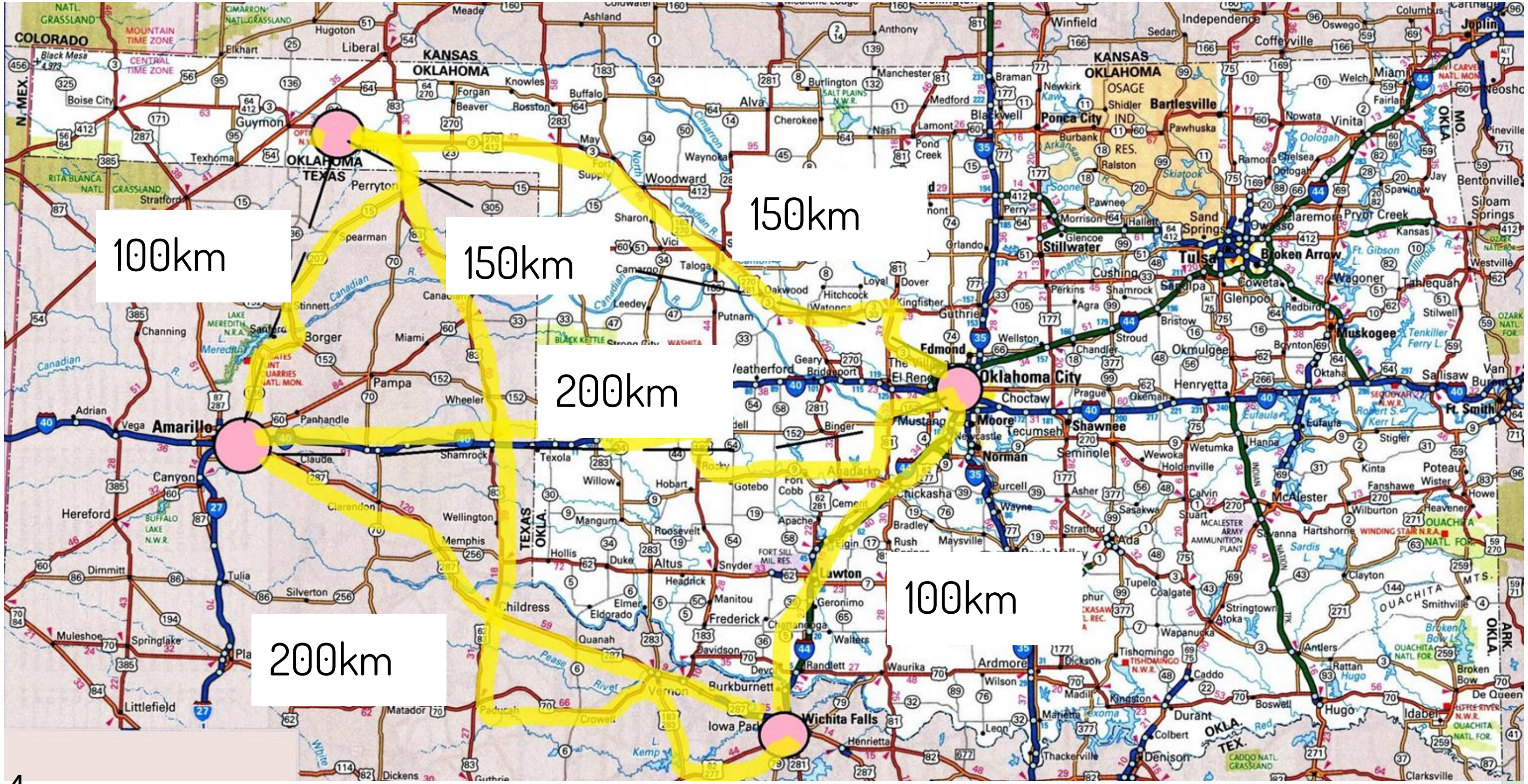
History of Algorithm

- Published by Robert Floyd 1962
- Same algorithm as Warshall's 1962
- 3 nested for loop version, Peter Ingerman

Algorithm Purpose: Floyd-Warshall

- Problem: You have multiple locations and you want to find the shortest distance between *every* specified location (All pairs, shortest path).
- Example: Calculating the distance on road maps between locations.
- A city represents a vertex, each road connecting the locations represents an edge and the numerical distance of the connecting roads represents the weight.

Algorithm Problem Example



Algorithm: Floyd-Warshall

- Finds the shortest path between all vertices in a weighted graph.
- Negative edge weights are permitted
- Negative *cycles* are not permitted.
- Solution is stored in an *nxn* adjacency matrix representation of the graph.

Floyd-Warshall Pseudocode

```
let V = # of vertices in graph
 let dist  = V * V solution array
 for each vertex(v)
     dist[v][v]=0 #Distance from itself is 0
 for each edge(u,v)
     #Edge weight placed use value inf if edge does not exist
     dist[u][v] = weight(u,v)
 for (k from 1 to V) #Each vertex picked as intermediate vertex
     for (i from 1 to V) #For the ith row
         for (j from 1 to V) #For the jth col
             if(dist[i][k]!=INF && dist[k][j]!=INF)  #Ignore if no edge
                 if(dist[i][j] > dist[i][k] + dist[k][j])
                     dist[i][j] = dist[i][k] + dist[k][j]
                 end if
             End if
 # If Distance in current slot is greater than the distance
 #it takes for the path to be completed by going through the current
 intermediate node, update the value
```

Floyd-Warshall Analysis:

- Runtime complexity: $\Theta(|V|^3)$
  - Every combination of edges is tested
  - 3 nested for loops
- $\Theta(|V|^2)$space complexity
- Modified to detect negative cycles

Implementation:

- Programmed in C++
- Graph[][]
  - <vector<vector<int>>
  - an adjacency matrix representation of a graph
- dist[][] - an n x n 2d matrix to store solutions

Problem Statement:

- Can Floyd-Warshall's algorithm be more efficient?
- Optimize to cut down on runtime

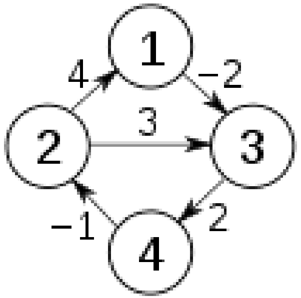
  Solution:

- Store calculation dist[i][k] + dist[k][j] in variable to avoid recalculation
- if (dist[i][k] != INF && dist[k][j] != INF) conditional
- FIRST had in j loop, moved to respective loop levels to reduce number of iterations
- Also solution Matrix[i][i] =0 (a diagonal always)

Floyd-Warshall: Demonstration

– Consider running the algorithm on the following graph
– Initial solution matrix state:

$k = 0$		j			
		1	**2**	**3**	**4**
i	**1**	0	∞	−2	∞
	2	4	0	3	∞
	3	∞	∞	0	2
	4	∞	−1	∞	0

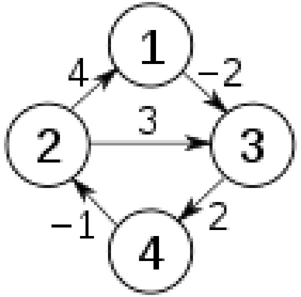


Process:

For your current *k* loop iteration compare value in $D[i][j]^{-1}$ to $D[i][k]^{-1} + D[k][j]^{-1}$ and choose the lesser value.

$k=0$		j			
		1	**2**	**3**	**4**
i	**1**	0	∞	−2	∞
	2	4	0	3	∞
	3	∞	∞	0	2
	4	∞	−1	∞	0

$k=1$		j			
		1	**2**	**3**	**4**
i	**1**	0	∞	−2	∞
	2	4	0	**2**	∞
	3	∞	∞	0	2
	4	∞	−1	∞	0



Example:

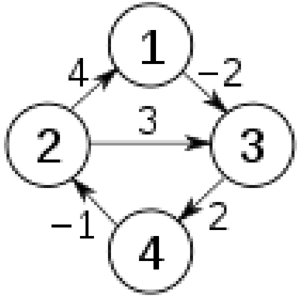
- To compute $D[4][1]^2$, you would first look at the value in $D[4][1]^1$, in this case it is infinity.
- Choose $\min(D[4][1]^1, D[4][2]^1 + D[2][1]^1)$.
- Update $D[4][1]^2$ to 3.

$k = 1$	j			
i	1	2	3	4
1	0	∞	−2	∞
2	4	0	2	∞
3	∞	∞	0	2
4	∞	−1	∞	0

$k = 2$	j			
i	1	2	3	4
1	0	∞	−2	∞
2	4	0	2	∞
3	∞	∞	0	2
4	3	−1	1	0

...

$k = 4$	j			
i	1	2	3	4
1	0	−1	−2	0
2	4	0	2	4
3	5	1	0	2
4	3	−1	1	0



Experimental Plan:

- Matrix filled with random numbers from -100 to 100 (except 0) with 25% or 75% chance of infinity
- Diagonals always filled with 0s
- Test F-W on the following sized matrices 100 times each for sparse graphs and dense graphs
  - 5x5, 100x100, 250x250, 500x500, 1000x1000
- Test F-W Opt on the same sized matrices 100 times each for sparse graphs and dense graphs

Optimization

```
for ( k = 0; k < inputSize; k++) {
for (i = 0; i < inputSize; i++)
{
  for (j =  0; j < inputSize; j++)
  {
    if (dist[i][k] != INF && dist[k][j] != INF)
    {
        int temp = dist[i][k] + dist[k][j];
              if (dist[i][j] == INF || temp < dist[i][j])
        {
      dist[i][j] = temp;
        }
    }
  }
}
```

```
void floydWarshallOptimized (vector<vector<int>> &Graph) {
  int dist[inputSize][inputSize], i, j, k;
  for (i = 0; i < inputSize; i++)
    for (j = 0; j < inputSize; j++)
      dist[i][j] = Graph[i][j];
    for ( k = 0; k < inputSize; k++){
      for ( i = 0; i < inputSize; i++ ) {
    if( dist[i][k] != INF ){ //IGNORE
    for ( j =  0; j < inputSize; j++ ) {
      /*Diagnol will always be 0,0 */
      if ( dist[k][j] != INF ||j==i) { //IGNORE
        int temp = dist[i][k] + dist[k][j];     //SAVE RECALCULATION
        if (dist[i][j] == INF || temp < dist[i][j]) {
          dist[i][j] = temp;
        }
      }
    }
   }
  }
 }
}
}
```

Output Example:

```
Graph:
0 32 17
INF 0 15
30 1 0

0 18 17
45 0 15
30 1 0
Average Time elapsed [STANDARD] 0.0000660
Average Time elapsed [OPTIMAL] 0.0000000
remote04:~/cs375algorithmPresent>
```

Dense Results:

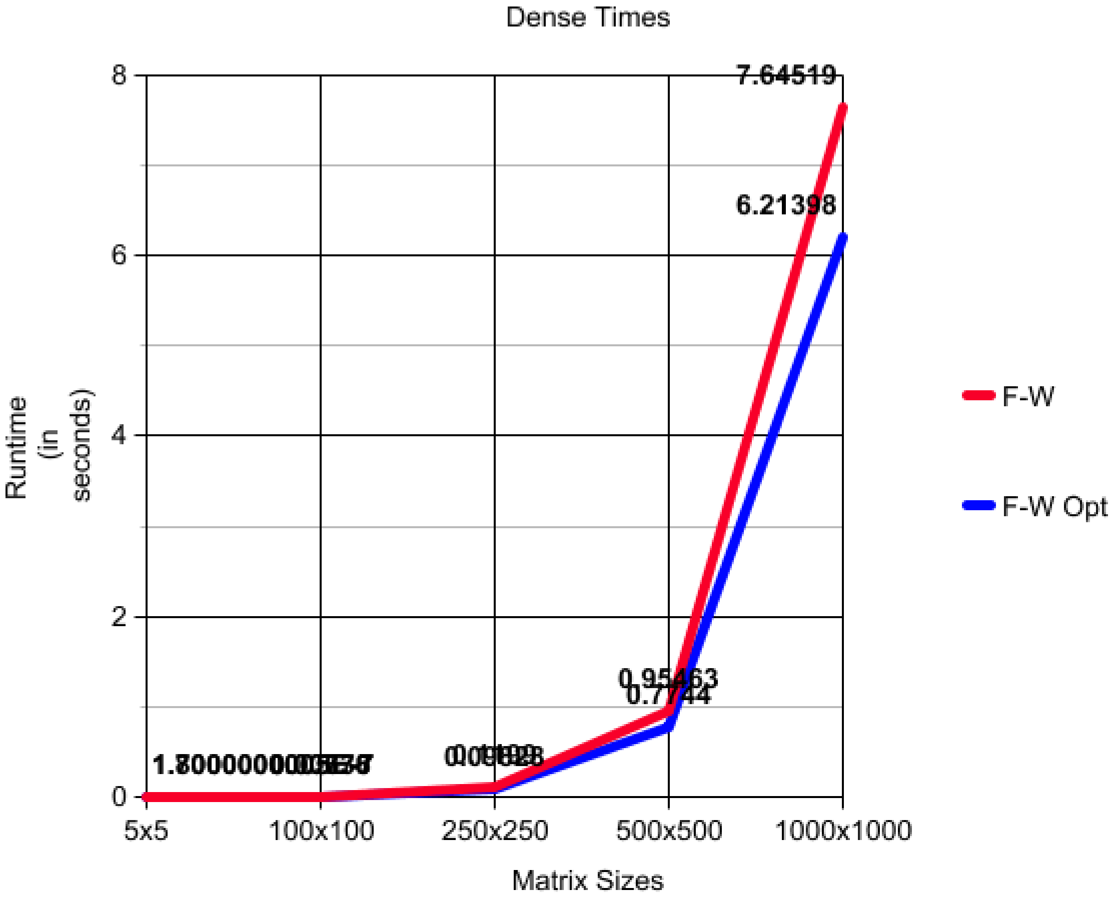
		Matrix Size				
		5x5	100x100	250x250	500x500	1000x1000
Runtime (in seconds)	F-W	0.0000018	0.00776	0.11990	0.95463	7.64519
	F-W Opt	0.0000017	0.00638	0.09828	0.77440	6.21398

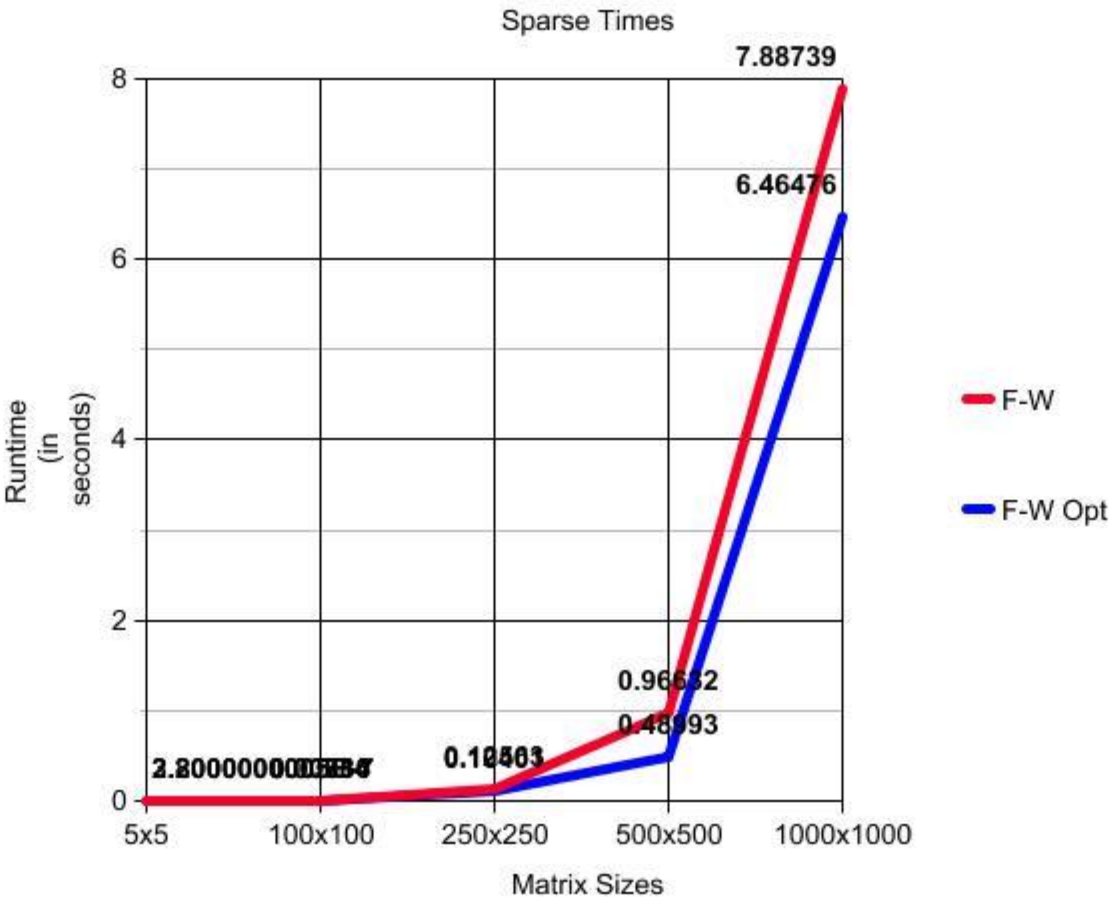
Sparse Results:

		Matrix Size				
		5x5	100x100	250x250	500x500	1000x1000
Runtime (in seconds)	F-W	0.0000032	0.00886	0.12563	0.96632	7.88739
	F-W Opt	0.0000028	0.00714	0.10401	0.48993	6.46476

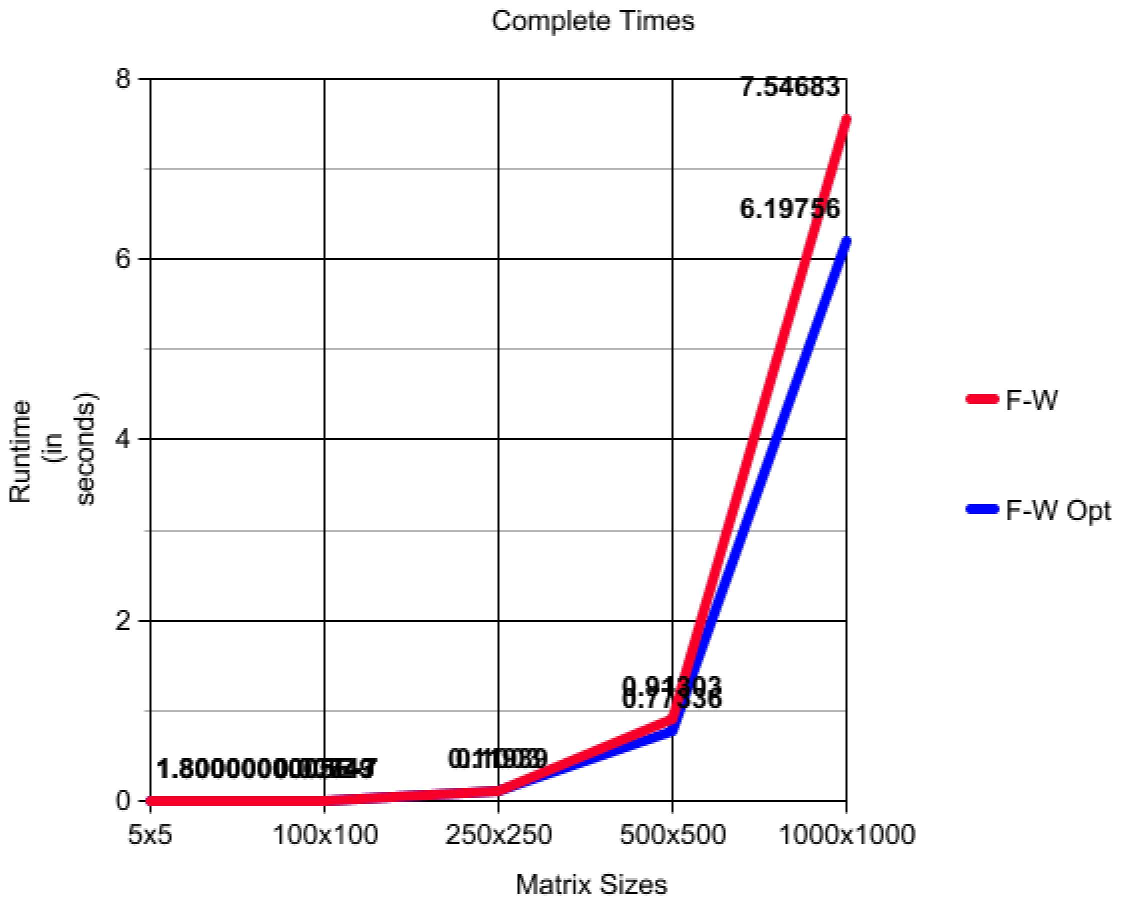
Complete Results:

		Matrix Size				
		5x5	100x100	250x250	500x500	1000x1000
Runtime (in seconds)	F-W	0.0000018	0.00749	0.11939	0.91303	7.54683
	F-W Opt	0.0000018	0.00643	0.10030	0.77336	6.19756


Dense Times
Runtime (in seconds)
8
6
4
2
0
7.64519
6.21398
0.95463
0.7744
5x5
100x100
250x250
500x500
1000x1000
Matrix Sizes
F-W
F-W Opt



Sparse Times
7.88739
6.46476
0.96632
0.48993
8
6
4
2
0
Runtime (in seconds)
5x5
100x100
250x250
500x500
1000x1000
Matrix Sizes
F-W
F-W Opt



Complete Times
Runtime (in seconds)
8
6
4
2
0
7.54683
6.19756
0.91393
0.77536
1.80000000000
5x5
100x100
250x250
500x500
1000x1000
Matrix Sizes
F-W
F-W Opt

Limitations and Future Work:

- Limited by computer speed and storage
- Implementation of recursive algorithm
- Comparison with more all-pair shortest path algorithms (i.e. fast matrix multiplication algorithms, Johnson's)
- Better data set

Concluding Remarks:

- Floyd Warshall is an all pair shortest path algorithm
- Useful yet expensive, better with optimization
- Works best for dense graphs
- Room for improvement

THANKS!

Any questions?