# A Distributed Algorithm for Minimizing Travel Time in Traffic Grids

Chinwei "Vic" Hu
Department of Electrical and Computer Engineering
The University of Texas at Austin
vic@cvhu.org

Saddam Quirrem
Department of Electrical and Computer Engineering
The University of Texas at Austin
s_quirrem@hotmail.com

***Abstract*—Modern traffic control systems often loop through a set of basic states, with little or no regard to the real-time traffic conditions and communications with neighboring systems. In this paper, we proposed a distributed algorithm, LAWQS, to design a communicative network of traffic management systems that are self-adaptive to real-time incoming traffic. We implemented a simulation framework to compare our algorithm against the traditional traffic light protocol, and was able to show a promising improvements in both vehicle traveling times and intersection traffic loads.**

## I. Introdution

The average commute time in the United States is 25.4 minutes [], and 80% of workers drive to work everyday. In 2011 along, American drivers spend 5.5 billion productive hours sitting in the traffic, costing the nation $121 billion dollars [].

Our current traffic infrastructure has been used for over a century, which wasn?t designed for the amount of traffic in the modern society. Most of the traffic lights and stop signs on the intersections today still use an inadaptable and synchronous architecture that waste a ton of unused time and space on the roads. Looking at the core concepts developed in communication networks, computer architecture, and distributed computing, many technologies such as virtual memory, pipelining, and network routing, are trying to solve very similar problems we are facing on the roads everyday. By introducing a few straightforward but yet powerful concepts, we wish to expand academia interests to solve these persisting problems in our current traffic control system.

In this paper, we introduced a simulation framework to model and simplify the sophisticated traffic systems, a set of measuring criteria to make quantitative analysis, a benchmark DUMMY algorithm, a Look-Ahead algorithm, a WeightedQ-Switcher algorithm, and a combination of these two algorithms. The rest of this paper is organized as following: section 2 covers the basic backgrounds, terminology, and notations; section 3 introduces the main algorithms; section 4 presents the experiment results, comparison analysis and discussions on our algorithms; section 5 makes the conclusion.

## II. Background

In this section, we will cover how the sophisticated traffic systems with a large degrees of uncertainty can be simplified and represented in a GridWorld model. Furthermore, we will formalize the problem we are trying to solve, go over some intuitions behind our solutions, and make clear definitions about the notations we will be using.

### A. The GridWorld Model

There are a great variation of possible configurations and protocols used in the real-world traffic systems. In our study, we only consider a two-lane, four-way configuration with all intersections perfectly aligned in a m-by-n GridWorld, as shown in Fig. 1

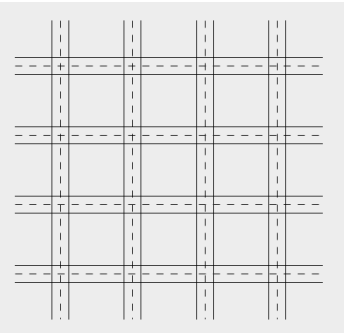

Fig. 1: a 4-by-4 GridWorld model

The intersections are modeled as a distributed system organized on a grid. We assume that each incoming vehicle is equipped with a communicating device such as a smartphone or a BlueTooth beacon, that is able to send requests to the upcoming intersection, and notify the driver when requests

are granted. We also assume that there is an application server installed at each intersection on the grid, which can communicate with its neighboring peers and respond to vehicles that have sent requests to it.

For instance, intersection at (0,0) refers to the southwestern corner of the grid, and intersection (n-1,m-1) refers to the northeastern corner of the grid. Each intersection acts as an independent node and has a direct two-way communication channel with its four neighboring intersections. This means intersection (x,y) can send and receive messages from intersections (x-1,y), (x+1,y), (x,y-1), and (x,y+1) if they exist in the grid. Each intersection server maintains four queues of vehicles (clients), and allows the head of each of these queues to pass if its requested direction matches the current state of the intersection server. The clients themselves perform two simple task. First they send a request to the facing intersection and wait until their request is acknowledge by an intersection server. When acknowledged, the vehicle client moves on to the next intersection, and continues until it reaches its destination.

### *B. The Intersection/Vehicle Architecture*

Each intersection server is essentially a finite state machine consisting of four states: `VERTICAL_STRAIGHT`, `VERTICAL_LEFT`, `HORIZONTAL_STRAIGHT`, and `HORIZONTAL_LEFT`, as shown in FIg. 10 `VERTICAL_*` states handle requests coming from the north and south directions, while `HORIZONTAL_*` states handle those from west and east. A request going straight and right is granted at `*_STRAIGHT` states, while a left-turn request are granted at `*_LEFT` states.

Each vehicle is assigned with a final destination and starting position, both of which are represented as an intersection on the grid. At each intersection, a vehicle can either take left, right, or go straight. There are two different kinds of vehicles. Ones that have a predetermined route to take, and ones that only specify the final destination and don?t care about the path. Note that the former can be easily mapped as a sequence of immediate short-term destinations, each spaced one block apart. Thus, the request protocol can be simply represented as the desired current destination node.

### *C. Problem Definition and Requirements*

Now that we have a clearly defined framework and a simplified model of our system, let?s revisit the problem and the requirements we are trying to achieve. In general, we want

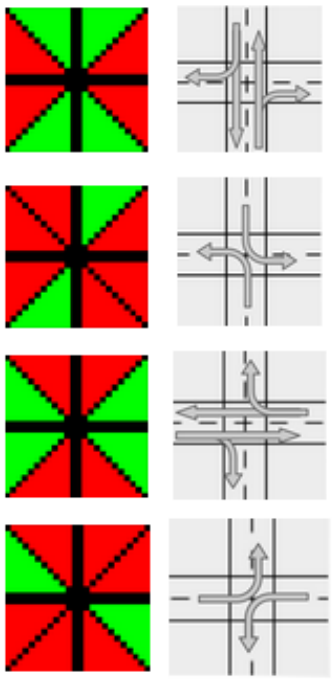

Fig. 2: the four intersection states

an algorithm that is distributed, scalable, flexible, adaptive, and secure; a system that can be reasonably integratable to our existing infrastructure.

***Problem Definition:*** *Given any grid configurations and vehicle routes, come up with a distributed algorithm to minimize time spent travelling and to even out intersection loads.*

In the real-world application, each vehicle ultimately only cares about how fast it can reach from the starting position to the final destination. In that regard, it?s obvious that everyone with the same destination is going to take the same optimal routes, creating high chance of gridlocks and leaving the rest of the grids unused. Therefore, we introduce a second objective function to even out the intersection loads to speed up the learning process.

**Requirements:**

- **Distributivity** To make the system architecture infinitely scalable, the algorithm should only communicate and utilize information of its own node and the neighboring nodes.
- **Safety** No conflicting traffic should be granted access to the intersection at anytime, any node.

- **Liveness** All requests should be granted within finite time.
- **Fairness** First come, first serve.

Similar to traditional distributed algorithms, these four fundamental requirements will be used to examine the correctness of our algorithms in the next section.

## III. Algorithms

In this section, we introduce four algorithms to be tested in our simulation framework: Dummy, Look-Ahead (LA), WeightedQ-Switcher (WQS), and LA-WQS.

### A. Intuitions

The Dummy algorithm represents the benchmark, which resembles the conventional traffic lights we see everyday on the street. It loops through a set of predefined states regardless of the real-time traffic. We use it as the baseline algorithm to demonstrate the effectiveness of our proposed ones. The Look-Ahead (LA) algorithm makes suggestions to incoming vehicles on the optimal action based on its neighborhood conditions. The vehicle only sends its desired final destination; the intersection server evaluates all the candidate immediate actions and responds to the request with the suggested action.

The WeightedQ-Switcher (WQS) algorithm makes decisions on state transition based on the weights of request queues. Each request sitting in the queue keeps a counter to represent how long it?s been waiting, and the sum of those counter values in a queue is the weight. When the intersection finishes a state operation, it picks the next state with the largest sum of weights.

Finally, the LA-WQS algorithm is simply a combination of both LA and WQS.

### B. Algorithms for Vehicles

There are two different kinds of vehicles: a PathVehicle that predetermines the route to take, and a DestVehicle that only specify the final destinations.

As shown in Algorithm. 1, the vehicle simply sends a request to the intersection server for each move, and waits until the request is granted by the server. When the request is granted, the server process calls the `receiveRequestOkay(nextPos)` function of the vehicle with the suggested next move, in which the vehicle moves on and continues the traversing procedure.

**Algorithm 1:** PathVehicle

```
Data: Start position and final destination
Result: Computes the destinationQueue and traverses
        through the path.
init destQ = generatePath(start, finalDest);
init currentPos = start;
init currentDest = destQ.pop();
while currentDest != null do
    sendRequest(currentPos);
    wait();
Function receiveRequestOkay(nextPos)
    Moves the vehicle to nextPos;
    if currentDest == currentPos then
        if destQ is not empty then
            currentDest = destQ.pop();
        else
            currentDest = null;
    notify();
Function generatePath(start, finalDest)
    init destQueue;
    currentPos = start;
    while currentPos != finalDestination do
        currentPos = randomWalk(currentPos, finalDest);
        destQueue.push(currentPos);
    return destQueue;
```

The DestVehicle employs a very similar model, except that there is no `generatePath` procedure anymore and `destQ` only contains the final destination.

### C. Algorithms for Intersections

Now we've covered the behavior of vehicle clients, let's look at algorithms followed by the server process at each intersection.

*1) Dummy Algorithm:* The Dummy algorithm is what we used as the baseline benchmark to measure the effectiveness of our proposed algorithms. It resembles the traditional traffic light system that loops through a predetermined set of states and durations, regardless of the real-time traffic conditions. As mentioned in Fig. 10, we modeled the traffic light system into a static finite state machine, consisting of `VERTICAL_STRAIGHT`, `VERTICAL_LEFT`, `HORIZONTAL_STRAIGHT`, and `HORIZONTAL_LEFT`. The state machine for the Dummy algorithm is illustrated in Fig. 3.

When a vehicle sends requests to an intersection, the server puts it into one of the four queues it keeps for each traffic direction: `north`, `south`, `easy`, and `west`. At each state,

the server examines the legal conditions of frontiers at the associated queues, and grants the requests if validated.

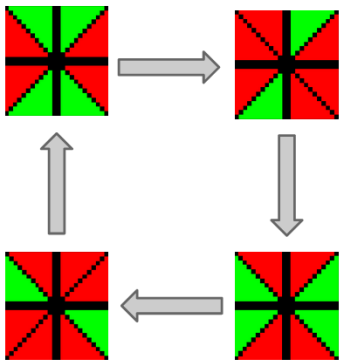

Fig. 3: a finite state machine for the Dummy algorithm

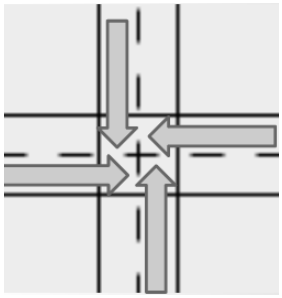

Fig. 4: Each intersection server keeps a queue to store requests coming from each direction.

**Algorithm 2:** Dummy

**Data**: Four requestQueues for each traffic direction.
**Result**: Grants requests based on the current state.
init requestsQMap as a hash map of request queues with direction as keys;
init FSM as the finite state machine defined in Fig. 3;
**while** *running* **do**
    currentState = getState(FSM);
    **for** *requestsQ in legal queues of currentState* **do**
        frontRequest = requestsQ.peek();
        **if** *frontRequest is legal* **then**
            grant frontRequest ;
            requestsQ.pop();

The legal states mentioned in Algorithm. 1 are determined according to the current state: requests queues of north and south are used when at `VERTICAL` states, while queues of east and west are used at `HORIZONTAL` states.

*2) Look-Ahead (LA) Algorithm:* Instead of granting incoming requests with a fixed path specified by the vehicle, the Look-Ahead algorithm makes suggestions on the optimal action based on the real-time traffic conditions of neighboring intersections.

In Fig. 5, a vehicle from the bottom right intersection is trying to reach the top left destination. Normally, an instance of PathVehicle has no prior knowledge on the traffic conditions, so it chooses both north and east with equal probability in a random walk path generating procedure. The Look-Ahead algorithm, on the other hand, sends messages to its neighboring intersection processes to get the queue sizes corresponding to the candidate actions, and suggests the vehicle to take the less-crowded route when granting the request.

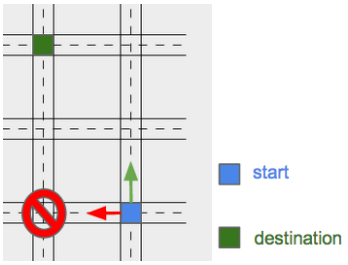


Fig. 5: Look-Ahead algorithm allows upcoming traffic to avoid neighboring gridlocks.

**Algorithm 3:** Look-Ahead (LA)

Everything is the same as in Algorithm. 2, except that when granting a request, the server makes action recommendation based on the following procedure:
**Function** `getOptimalAction(`*destination*`)`
    init minSize = -∞;
    init actionOptimal = null;
    **for** *action in candidateActions* **do**
        **if** *queue size with action ¡ minSize* **then**
            minSize = queue.size();
            actionOpt = action;
    **return** actionOpt;

The concepts behind Look-Ahead is very similar to Dijkstra's shortest path with horizontal of size one, and is fairly simple to implement. For each request, there are one request message, two to four look-ahead messages, and one response message. In practice, the neighboring queue size data could be cached in the current intersection server, and updates could be accomplished by a publisher/subscriber paradigm. In such cases, the message complexity can be reduced down to size two, which is the same as in Dummy.

One of the biggest disadvantage in both Dummy and Look-Ahead is that they both utilizes static state transitions,

which wastes a lot of unused cycles and resources as in most synchronous systems. Next, we will talk about how the WeightedQ-Switcher algorithm addresses exactly this problem.

*3) WeightedQ-Switcher (WQS) Algorithm:* In our analysis for the Dummy and Look-Ahead algorithms, we discussed how a hard-coded state machine suffers from wasted resources and computation cycles, which could be resolved by designing an adaptive system that makes state transition decisions based on the real-time traffic. In this section, we propose the WeightedQ-Switcher algorithm to give a simple example on how that could be done in practice.

As shown in Fig. 6, the state machine presented in Fig. 3 is separated into a dynamic, two-state machine. The new `VERTICAL` state consists of both `VERTICAL_STRAIGHT` and `VERTICAL_LEFT`, which takes care of the traffic from north and south before either switching to `HORIZONTAL` or staying in `VERTICAL`, and similarly for the transitions in `HORIZONTAL`.

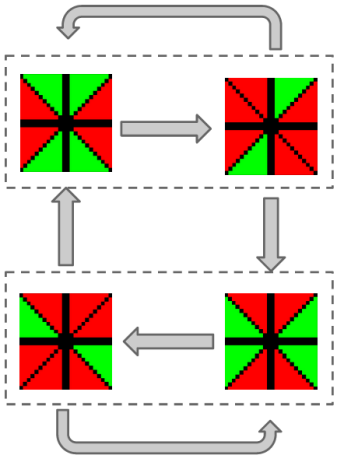

Fig. 6: A dynamic state machine of WQS

In Fig. 7, we see how WQS decides which state to go next based on `w(VRTL)`, the weights of vertical queues, and `w(HRZL)`, the weights of horizontal queues. Note that if we naively define queue weights as the total number of requests, we may easily run into cases where requests sitting in certain queues are never processed, and hence suffer from starvation. Therefore, we incorporate a counter parameter for each waiting elements in queues, which represents how many rounds it's been waiting for the server process.

More details on this procedure are specified in Algorithm. 4, in which we omitted parts that are duplicated from previous algorithms.

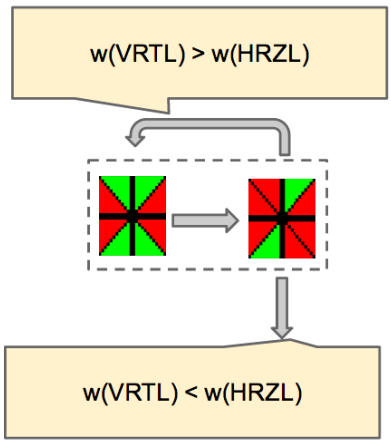


Fig. 7: State transitions in WQS is now based on the queue weights.

**Algorithm 4:** WeightedQ-Switcher (WQS)

Everything is the same as in Algorithm. 2, except that when transitioning to the next state, the intersection server calls the following procedure:

```
Function getNextWQSState()
    init nextState = null;
    init weightVertical = getWeight(north) +
    getWeight(south);
    init weightHorizontal = getWeight(east) +
    getWeight(west);
    if weightVertical ¿ weightHorizontal then
        nextState = VERTICAL;
    else if weightVertical ¡ weightHorizontal then
        nextState = HORIZONTAL;
    else
        nextState = random(VERTICAL,
        HORIZONTAL);
    return nextState;
Function getWeight(direction)
    init weight = 0;
    for request in requestsQMap.get(direction do
        weight += request.getCounter();
    return weight;
```

In the real-world analogy, we can think of WQS as a mean to allow queues that are large and have been waiting longer than others to pass with privilege. On the other hand, if there is no traffic on the other direction, one would never have to stop unnecessarily.

*4) LA-WQS Algorithm:* As the name suggests, LA-WQS is a combination of both the afore-mentioned Look-Ahead and WeightedQ-Switcher algorithms.

### D. Correctness Analysis

- **Distributivity** To make the system architecture infinitely scalable, the algorithm should only communicate and utilize information of its own node and the neighboring nodes.
- **Safety** No conflicting traffic should be granted access to the intersection at anytime, any node.
- **Liveness** All requests should be granted within finite time.
- **Fairness** First come, first serve.

### E. Complexity Analysis

## IV. Experiments

### A. Testing Methodology

The average velocity, measured as the number of nodes traveled divided by the time taken for a vehicle to reach its destination, is used to compare the performance of each algorithm. The algorithms will be tested with various grid size and the number of vehicles. A higher average velocity is would indicate a higher throughput.

### B. Results and Discussion

The results show that making use of a distributed algorithm will result in a more even distribution of vehicles across the grid and greater throughput. Some basic runtime analysis was performed on four algorithms with grid sizes ranging from 4x4 to 8x8 and either 256 or 512 vehicles.

| DUMMY-256 | 4 | 6 | 8 | LA-256 | 4 | 6 | 8 |
|---|---|---|---|---|---|---|---|
| 4 | 39013 | 39253 | 32550 | 4 | 21422 | 28676 | 25200 |
| 6 | 41309 | 43304 | 44127 | 6 | 18072 | 15129 | 19498 |
| 8 | 36259 | 45427 | 41077 | 8 | 14837 | 18586 | 16586 |
| DUMMY-512 | 4 | 6 | 8 | LA-512 | 4 | 6 | 8 |
| 4 | 60226 | 68104 | 59637 | 4 | 39244 | 42629 | 40099 |
| 6 | 57216 | 50822 | 57736 | 6 | 28816 | 25745 | 22987 |
| 8 | 50831 | 59116 | 64390 | 8 | 15670 | 23909 | 28594 |
| WQS-256 | 4 | 6 | 8 | LAWQS-256 | 4 | 6 | 8 |
| 4 | 52667 | 66518 | 50349 | 4 | 30896 | 32157 | 36283 |
| 6 | 48257 | 65807 | 54196 | 6 | 30103 | 33735 | 26485 |
| 8 | 45703 | 63645 | 58044 | 8 | 30606 | 29260 | 32657 |
| WQS-512 | 4 | 6 | 8 | LAWQS-512 | 4 | 6 | 8 |
| 4 | 111592 | 71073 | 81941 | 4 | 51589 | 43182 | 34841 |
| 6 | 88475 | 79650 | 81150 | 6 | 41061 | 41002 | 44995 |
| 8 | 86961 | 80793 | 84447 | 8 | 42095 | 42671 | 38542 |

Fig. 8: the four intersection states

These results cannot be used to evaluate the throughput based on the size of the grid, because much of the runtime is taken up by a few vehicles which have fallen behind. However, what can be taken from these results is which algorithms will generally outperform the other. From these results we can see that the top performing algorithms are Lookahead, LA-WQS, Dummy, and WeightedQ-Switcher. It seems that, contrary to what was initially believed, switching states based on the ?weights? of the lanes is not beneficial. The explanation for this is that if the intersection server spends it time servicing only one lane, then the other three lanes will eventually get clogged. This creates a cycle where the intersection server would keep serving heavier lanes of increasing weight until no more vehicles are coming in.

The combination of the lookahead algorithm and the weightedQ-switcher actually managed to outperform the dummy algorithm. This shows that just by adding a distributed algorithm to a slower non-distributed algorithm, we were able to outperform the simplest implementation. An analysis of the workload shows how traits of our WQS and LA algorithm were inherited by the LA-WQS algorithm.

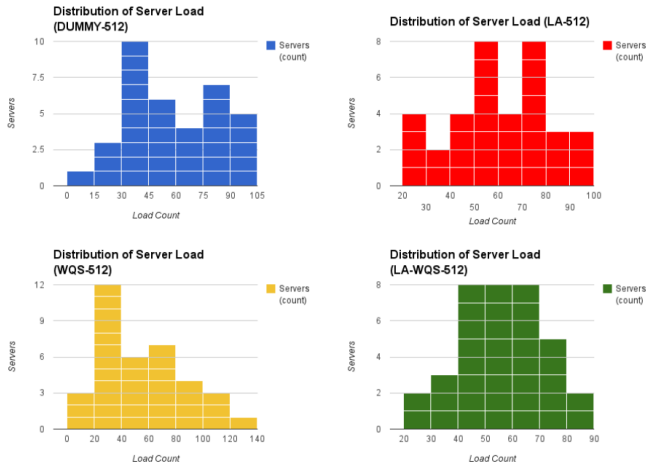


Fig. 9: the four intersection states

The above figures are histograms displaying the number servers in each workload bucket, where each bucket is identified by how many vehicles pass through each server. An ideal algorithm would have a normal distribution for the server load with as small a standard deviation as possible. The lookahead algorithm shows a somewhat normal distribution, and this trait is inherited by the LA-WQS algorithm?s distribution. On the other hand, using the dummy algorithm resulted in a greater number of servers taking on much higher loads. The WQS algorithm seems to underutilize many server.

The motion of the vehicles provides the overall throughput of the system. The vehicles rate of motion is calculated as the average number of logical clock cycles it takes to move to a single intersection. A lower number of logical clock cycles per move would be indicative of a higher throughput. The four

algorithms were tested with the same configuration to extract the server workload data used to extract the vehicle motion data.

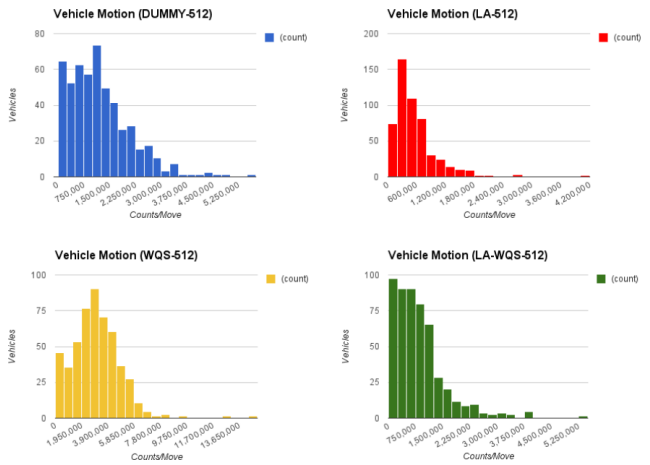


Fig. 10: the four intersection states

The distributions which were more skewed to the left (less counts/move) had a higher throughput. The two lookahead algorithms had the highest throughput and were the most skewed leftward. The WQS algorithm had very poor throughput and is shown to have only held back the vehicles.

## V. Conclusion

The use of a distributed algorithm has greatly increased both throughput and balance in the server load. The lookahead algorithm not only outperformed the other algorithm, but its addition to the slowest non-distributed algorithm resulted in it becoming the second best performing algorithm. What this shows is that it is not enough to vehicles to their next intersection as quickly as possible and that the workload must be balanced across the grid. The lookahead algorithms had the most balanced workloads, as evidenced by the normal distribution of their workloads.

The lookahead algorithm only has an intersection server send a query to its immediate neighbors. It is likely that by query even further for the shortest path would normalize the server workload even further, but at the cost of greater message complexity. Implementing Dijkstra?s algorithm for finding the shortest path could prove costly for larger grids. However, there could be some ideal level of broadcast-convergecast messages that could used depending on the size of the grid.

## VI. Future Improvements

## VII. Acknowledgements

## References


[1] S Census
[2] AMU paper
[3] andadian paper
[4] Dijkstra's shortest path
[5] Forney Jr, G. D. (1973). The viterbi algorithm. Proceedings of the IEEE, 61(3), 268-278.