Gridlock-free Autonomous Parking Lots for Autonomous Vehicles

Tsz-Chiu Au[1]

***Abstract*— Many cities suffer from a shortage of parking spaces. Research in high density parking (HDP) focuses on how to increase the capacity of parking lots by allowing vehicles to block each other but temporarily give way to other vehicles by driving autonomously upon request. Previous works on HDP did not consider mixing different parking strategies and ignored the possibility of gridlock when multiple vehicles move simultaneously. In this paper, we describe the design of *autonomous parking lots*, which allows the deployment of different parking strategies in different regions in a parking lot. We present algorithms for checking whether adding a vehicle to an autonomous parking lot can lead to gridlock. Our simulation shows that autonomous parking lots can hold 60% more vehicles given the same amount of space.**

I. INTRODUCTION

In many cities, finding a parking space is difficult due to the limited amount of parking space. Recent research on high density parking (HDP) focuses on utilizing autonomous driving to greatly boost the capacity of parking lots [1], [2], [3], [4]. The design of conventional parking lots reserves more than half of the space for driveways and sidewalks [3], [5]. HDP reclaims these spaces by allowing vehicles to park in the driveways or reducing the number of driveways by putting vehicles close to each other. An autonomous vehicle in the parking lot can be asked to move if it blocks another vehicle that needs to leave the parking lot. HDP can increase the capacity of a parking lot by an average of 62% according to one study [6]. HDP can be realized by fully automated driverless SAE Level 4 parking functions recently introduced by major car manufacturers.

The early high-density parking lot design put vehicles in queues so that vehicles circle round in a parking lot [7]. However, most recent works on HDP prefer putting vehicles in stacks instead [1], [8], [3], [6], [4]. All of these works have not considered mixing both queues and stacks in a parking lot design. Let us take a look at the parking lot in Fig. 1, which allocates queues (the purple areas) in the middle area and stacks (the red areas) near the upper and lower boundaries. It is hard to tell whether replacing the queues with stacks is a better choice in this parking lot, as it appears that vehicles in the queues can get in and out of the parking lot easily given the locations of the entries and the exits. Hence, we should not immediately conclude that stacks are better than queues without considering the geometrical shape of parking lots and the locations of entries and exits.

In this paper, we present a high-density parking lot design called *autonomous parking lots* (*APLs*) in which queues and stacks can coexist. The idea is that a parking lot is partitioned

[1]Department of Computer Science and Engineering, Ulsan National Institute of Science and Technology, South Korea. chiu@unist.ac.kr

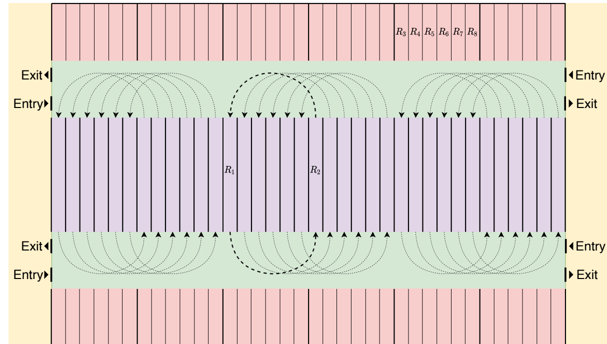

Fig. 1: The layout of an autonomous parking lot.

into several regions. Each region can be an empty space, a stack, a queue, or other structures. Different regions are managed by different management agents separately. The management agents use the empty space (the green areas in Fig. 1) to send vehicles in and out of their regions and the parking lot. The traffic in an empty space is managed by a reservation system similar to Autonomous Intersection Management (AIM) [9], such that vehicles have to reserve the spaces on their trajectories in the empty space before they can enter the empty space.

Mixing both queues and stacks can potentially lead to *gridlock*—some vehicles are blocked by other vehicles forever and cannot leave the parking lot. Moreover, unlike most previous works, we do not put every vehicle in a parking space of a predefined size (e.g., $5m \times 2m$), since this one-size-fit-all scheme could waste a lot of space if vehicles have different sizes. Gridlock can occur inside queues and stacks if vehicles use different amounts of parking space, as we shall discuss in Sec. IV. This paper presents how to prevent gridlock even if we mix queues, stacks, or any other parking strategy in an autonomous parking lot. In summary, the contributions of this paper are:

- We present the design of autonomous parking lots, including the reservation system and the communication protocol between the management agents.
- We analyze the sufficient conditions under which gridlock would not occur in APLs and present algorithms for gridlock detection and prevention.
- We implemented a simulator of autonomous parking lots and conducted experiments to compare conventional parking lots with autonomous parking lots.

This paper is organized as follows. After the related work section in Sec. II, we define all components in an APL in Sec. III and analyze the conditions for gridlock-free APLs in Sec. IV. Finally, we present our experimental results in Sec. V and conclude this paper in Sec. VI.

II. RELATED WORK

Automatic parking has already been a commercial feature in some luxury vehicles. Thanks to deep learning's superior performance in computer vision tasks such as parking scene recognition, automatic parking has become much more reliable [10]. In 2019, Tesla rolled out a feature called Smart Summon[2], which enables a customer to summon their car from across a parking lot remotely. Automated valet parking (AVP) allows passengers to leave the car in a drop-off zone and let the car park itself. Last year, Bosch, Ford[3], and Mercedes-Benz[4] demonstrated the AVP feature of their vehicles. [11] is a recent survey of AVP.

As the AVP technology becomes mature, we are poised to tackle parking problems in urban transportation. High density parking (HDP) addresses the shortage of parking space by utilizing AVP in parking lots and allowing vehicles blocks each other temporarily. [7] is an early study of HDP that puts vehicles in parallel rows such that vehicles can cycle around. To our knowledge, it is the only work that organizes vehicles as queues in HDP. However, the subsequent HDP research dismissed queue structures and prefers stacking vehicles in a first-in-last-out manner. Timpner et al. [1] optimized parking space by k-stacks, which put multiple vehicles in a perpendicular parking slot. D'Orey et al. [8] proposed an automated parking model in which parking areas are a group of stacks, and vehicles have to utilize the buffer area to move between the stacks. They also devised some planning strategies for controlling vehicles in the parking lot [3]. Azevedo et al. [4] thoroughly evaluated the high-density parking lots in [8] with different reallocation strategies and spatial configurations. Banzhaf et al. [2] presented a novel approach called k-deques that parks vehicles in driving lanes. Nourinejad et al. [6] presented another stack-based approach in which a parking lot is divided into several islands, each contains multiple stacks.

Apart from stacks and queues, other parking strategies do exist. Zips et al. [12] increased the capacity of parking lots by up to 12% by varying parking spots' width. Our APL can also take advantage of different vehicles' width by having stacks and queues of different widths in a parking lot.

III. AUTONOMOUS PARKING LOTS

This section defines all components in APLs, including the regional managers, the gate managers, the reservation handlers, and the communication protocols.

A. Regions and Managers

A *region* R is a closed, path-connected, non-empty space on a 2D-plane in $\Re^2$, with or without "holes" (i.e., the region may not be simply connected). R can be a non-rectangular region with obstacles such as columns or light poles. The areas occupied by the obstacles are not part of R. ∂R denotes the outer boundary of R, which excludes the boundaries of the obstacles inside R.

Every region R has at least one *entry* and at least one *exit*, which are line segments on the boundary ∂R. A vehicle can enter R via an entry and leave R via an exit. Two regions are *connected* or *adjacent* if (1) their boundaries overlap each other, and (2) they share entries or exits on their overlapping boundary. If R_1 and R_2 are connected, an entry of R_1 on their shared boundary is an exit of R_2, and vice versa.

There are two types of regions in a APL: *managed regions* and *shared regions*. Vehicles are stored in managed regions only. In Fig. 1, the red regions and the purple regions are managed regions. The green regions are shared regions, which are driveways that allow vehicles to get in and out of the managed regions and the parking lot. Vehicles are forbidden to stop inside a shared region. A managed region can only connect to shared regions but not another managed region. A shared region can connect to managed regions or *external regions*—the area outside the parking lot (the yellow regions in Fig. 1)—but not another shared region.

Managed regions are controlled by *regional managers*, which implement a *parking strategy* that decides how vehicles are parked inside the managed regions. Note that a regional manager can manage one or more managed regions, but a managed region can only be managed by one regional manager. Let $\mathcal{R}(\pi)$ be the set of managed regions managed by a regional manager π. Vehicles in external regions are managed by *gate managers*, which control when a vehicle can enter or leave the parking lot. There is one gate manager for each external region.

B. Queue Groups and Stack Groups

Queues and stacks are the two parking strategies proposed in the previous works. Both of them are suitable for lining up vehicles with nonholonomic motions. Queues arrange vehicles in a first-in-first-out (FIFO) manner such that vehicles enter at one end and leave from the other end. The drawback of queues is that the two ends of a queue cannot be blocked by walls or obstacles. Stacks organize vehicles in a first-in-last-out (FILO) manner and require one open end only.

We combine two or more queues to form a *queue group*. For instance, in Fig. 1, R_1 and R_2 form a queue group. Let $\mathcal{R}(\pi) = \langle R_1, R_2, \ldots, R_n \rangle$ be the sequence of regions managed by a queue manager π, where R_1 and R_n are the head and the tail of the queue group, respectively. Vehicles in a queue group can move from one region to another, such that the first vehicle ν in R_i can move to the end of R_{i-1} (or R_n if $i = 1$), for $1 \leq i \leq n$. In a *rotation operation*, the first vehicle in R_1 moves to the end of R_n. The success of a rotation operation depends on the movement of other vehicles in the queue group. A vehicle ν cannot leave a queue group if it is not the first vehicle of any queue R_i. Hence, we need to apply the rotation operation to relocate ν such that it becomes the first vehicle of some queue. The regional manager of a queue group is called a *queue manager*.

Similarly, we can combine several stacks to form a *stack group*. In Fig. 1, $\{R_3, R_4, \ldots, R_8\}$ is a stack group. In a

[2] https://www.tesla.com/support/autopilot

[3] https://www.caranddriver.com/news/a33808473/ford-bosch-automated-parking-pilot

[4] https://www.daimler.com/innovation/case/autonomous/driverless-parking.html

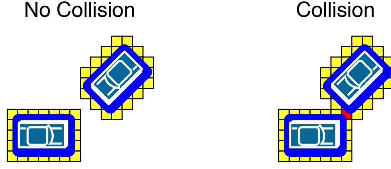

Fig. 2: Collision detection via a reservation system. The blue areas around the vehicles are *safety buffers* in which no other vehicle shall present at any time.

relocation operation, the stack manager asks a vehicle to move to another stack to give way to another vehicle below it. The regional manager of a stack group is called a *stack manager*.

Unlike most existing works on HDP, APL does not have a fixed-size parking space for every vehicle. Vehicles of different lengths can park "bumper-to-bumper" in the same managed region, as long as they maintain a minimum distance called the *car gap* (i.e., 10 cm) between them.

C. Reservation Handlers for Shared Regions

Vehicles enter or leave managed regions via shared regions. As many vehicles can use a shared region at the same time, we need a way to coordinate vehicles in shared regions. APLs use a reservation system similar to Autonomous Intersection Management (AIM) [9] for vehicle coordination and collision avoidance in shared regions. In AIM, an autonomous vehicle ν has to reserve a block of space-time in an intersection before entering the intersection. The reservation is administrated by a *reservation handler*, which divides the area in an intersection into a grid of *tiles* (i.e. grid cells). When ν approaches an intersection, the IM uses the data in the reservation request regarding the time and velocity of arrival, vehicle size, etc., to simulate the intended trajectory of ν across the intersection. At each simulated time step, the IM determines which tiles will be occupied by ν and ν's safety buffer. If these tiles have no conflict with the reserved tiles, the reservation succeeds and the tiles will be reserved for ν; otherwise, the reservation request will be rejected (see Fig. 2). After that, ν can use the reserved tiles to pass through an intersection safely. We also adopt this reservation system for managing vehicles in shared regions.

Every shared region has a reservation system controlled by a *reservation handler* (RH). Before a manager sends a vehicle to a shared region R, it submits a *request* to the RH of R. A request is a tuple (t, ν, π, T), where ν is a vehicle, t is the time ν plans to enter R, π is the manager, and $T = \langle(t_1, \tau_1), (t_2, \tau_2), \ldots, (t_k, \tau_k)\rangle$ is the list of space-time tiles, meaning that ν needs to reserve τ_j at time t_j. At every time step, a RH collects a list of requests from the managers of the adjacent regions. The RH processes the requests according to Algorithm 1. The RH stores the requests in a list of request queues: $Q_{t_0}, Q_{t_0+1}, \ldots, Q_{t_0+T}$, where t_0 is the current time and T is the time horizon the RH would consider. Then the RH looks at Q_{t_0} and accept some requests in Q_{t_0} and reject the remaining requests. First, the RH calculates the priority

Algorithm 1 The reservation handler of a shared region R.

1: **procedure ReservationHandler**(R)
2:   **while** True **do**
3:     Let t_0 be the current time step
4:     Receive requests from the managers of the adjacent regions
5:     Store the requests in the request queues: $Q_{t_0}, \ldots, Q_{t_0+T}$
6:     **while** Q_{t_0} is not empty **do**
7:       Calculate the priority $f(r)$ for all request r in Q_{t_0}
8:       Randomly choose $r = (t_0, \nu, \pi, T) \in Q_{t_0}$ based on $f(r)$
9:       **if** some tiles in T have been reserved **then** reject r
10:       **else**
11:         Reserve the tiles in T for ν and accept r
12:         Remove requests that involve ν in the request queues

Algorithm 2 The simple regional manger π.

1: **procedure SimpleRegionalManager**(π)
2:   **while** True **do**
3:     **for all** request $r^{\text{park}} = (t, \nu)$ for parking in $\mathcal{R}(\pi)$ **do**
4:       **if** there is enough space in $\mathcal{R}(\pi)$ to hold ν at time t **then**
5:         **if** gridlock check ok **then** accept r^{park} **else** reject r^{park}
6:       **else**
7:         Reject r^{park}
8:     **for all** $r^{\text{unpark}} = (\nu, e)$ for leaving the parking lot **do**
9:       Mark ν as "leaving from exit e"
10:     **for all** vehicle ν marked as "leaving from exit e" **do**
11:       Move ν to an exit e' of $\mathcal{R}(\pi)$ that can reach e
12:       **if** ν has arrived at e' **then**
13:         Send a request r^{reserve} to the RH of the shared region R connecting e' to e, where t_0 is the current time.
14:         Send $r^{\text{exit}} = (\nu, e, t')$ to the gate manager of e
15:         **if** both r^{reserve} and r^{exit} are accepted **then**
16:           ν leaves $\mathcal{R}(\pi)$ and enters R.

of every request $r = (t, \nu, \pi, T)$ in Q_{t_0} using this equation:

$$f(r) = w(r) \times e^{t_{\text{last}}(\pi)}, \tag{1}$$

where (1) $t_{\text{last}}(\pi)$ is the time passed since the manager π's last request was accepted by the RH, and (2) the value of $w(r)$ depends on the type of request r. The requests for the vehicles leaving the parking lot will have the largest value of $w(r)$, whereas the requests for the vehicles entering the parking lot will have the smallest value of w. As discussed in [13], the use of the priority function can prevent some vehicles from failing to enter an intersection indefinitely. The same is true for APLs as long as the managers keep sending requests to the RH after rejection.

After computing the priority of the requests in Q_{t_0}, the RH randomly select a request $r = (t, \nu, M, T) \in Q_{t_0}$, and the chance of choosing a request is proportional to the priority. The RH rejects r if any tile in T has been reserved previously; otherwise, the RH will accept r and let ν enter the shared region at time t by reserving the tiles in T for ν. Since ν has been accepted, the RH will remove all requests that involve ν from the request queues. The above steps repeat until all requests in Q_{t_0} have been either accepted or rejected.

D. Communication Protocols between Managers

Requests received by RHs are generated by gate managers or regional managers, which also accept requests from other

Algorithm 3 The gate manager for an external region R^{ext}.

```
1: procedure GateManager(R^ext)
2:   while True do
3:     for all request r^enter = (ν, e) to enter the parking lot do
4:       D := FindManagedRegionEntry(ν, e, R^ext)
5:       if D is empty then reject r^enter
6:       else
7:         for all (e, R, e', π) in D in a random order do
8:           Let S_1 = {(e, R, e')} be the driving scenario in R
9:           Let S_2 be the set of required driving scenarios in M
10:          Ask ν whether it can handle all scenarios in S_1 ∪ S_2.
11:          if ν can handle all scenarios in S_1 ∪ S_2 then
12:            Compute the set T of tiles going from e to e'
13:            Let t' be the last time step of T
14:            Send r_1 = (t, ν, π, T) to the RH of R
15:            Send r_2 = (t', ν) to the regional manager of e'
16:            if both r_1 and r_2 are accepted then
17:              Accept r^enter and ν enters the parking lot to e'
18:              break
19:        Reject r^enter      // ν cannot enter at this time
20:    for all request r^exit = (ν, e, t) to leave the parking lot do
21:      if the exit e is not blocked at time t then accept r^exit
22:      else reject r^exit
23: procedure FindManagedRegionEntry(ν, e, R^ext)
24:   Let R be the shared region adjacent to R^ext through entry e
25:   D := {}
26:   for all regional manager π adjacent to R do
27:     if ν can fit inside R(π) according to the spec of ν then
28:       for all entry e' of R(π) that is also on ∂R do
29:         D := D ∪ {(e, R, e', π)}
30:   return D
```

managers. In this section, we describe the communication protocols between different managers and RHs.

Algorithm 2 describes the behavior of regional managers such as queue managers and stack managers. These managers can receive a request $r^{\text{park}} = (t, \nu)$ to park a vehicle ν at time t from other managers. They can also receive a request $r^{\text{unpark}} = (\nu, e)$ to leave the parking lot via exit e from a vehicle in the regions they manage. These managers decide whether they accept r^{park} according to the space availability and some gridlock conditions that will be described in Sec. IV. r^{unpark} is handled by moving the vehicle ν to an exit e' of the managed regions such that ν can reach the exit e via a shared region R. Then the regional manager sends a request to the RH of R repeatedly and sends a request r^{exit} to the gate manager of e repeatedly until ν can enter R so as to leave the parking lot through e.

Gate managers are more complicated than the simple regional managers since they have to decide which managed region a vehicle should park. Algorithm 3 describes how a gate manager works. When a vehicle ν arrives at an entry e, it will repeatedly send a request r^{enter} to the gate manager of e. The gate manager will then identify the set of regional managers who can hold ν according to the vehicle's specification (Line 27). At this point, the regional managers only concern with the physical dimension of ν (e.g., whether ν is too wide to fit in a stack). Besides, ν has to be able to reach one of these managed regions' entries from e. For each reachable

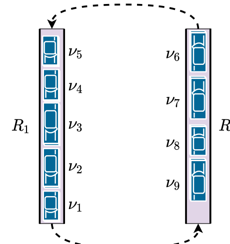

Fig. 3: A gridlock in a queue group.

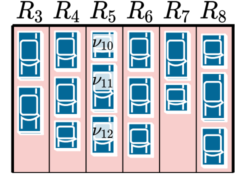

Fig. 4: The lack of space for relocation in a stack group.

regional manager π found by **FindManagedRegionEntry**(), the gate manager will compile a list S of driving scenarios in $\mathcal{R}(\pi)$ as well as the driving scenario in the shared region R through which ν goes from e to an entry e' of $\mathcal{R}(\pi)$. Each driving scenario consists of a start pose and an end pose of a vehicle, and some physical constraints such as the boundary of the driveway, the speed limit, etc. The gate managers will send the list of driving scenarios to ν and ask whether it can go to the end poses from the start poses without violating any constraints in the scenarios (Line 10). If ν can handle all scenarios, the gate manager computes the set T of tiles on ν's trajectory and send a request to π and a request to the RH of R. If both requests are accepted, ν enters the parking lot. A gate manager always accepts an exit request $r^{\text{exit}} = (\nu, e, t)$ unless the exit e will be blocked at time t.

IV. GRIDLOCK PREVENTION

In this section, we present the condition under which no vehicle will get stuck in an APL forever.

A. 1-Cyclability of Queue Groups

Fig. 3 shows a gridlock in a queue group. Suppose ν_3 needs to leave the parking lot. ν_1 and ν_2 have to move to R_2 so that ν_3 can leave. First, ν_1 cannot move to R_2 since there is not enough room in R_2. Second, if we move ν_1 and ν_6 *simultaneously* such that R_2 has enough room to hold ν_1 after ν_6 leaves R_2, the total length of ν_2, ν_3, ν_4, ν_5 and ν_6, together with the car gaps between them, exceeds the length of R_1, and ν_6 cannot join R_1. Therefore, ν_3 can never leave the queue group and gridlock occurs. To prevent such gridlock, the queue manager has to check some conditions before adding a vehicle to its regions. Here we present two such conditions, both of them can guarantee gridlock-free of queue groups. But only one of them constitutes the sufficient condition of gridlock-free APLs in Sec IV-C.

Let $\mathcal{V}$ be a vehicle sequence $\langle \nu_1, \nu_2, \ldots, \nu_n \rangle$ on a queue or a stack. We define the length of $\mathcal{V}$ as $||\mathcal{V}|| = \left\{\sum_{\nu \in \mathcal{V}} |\mathcal{V}|\right\} + (|\mathcal{V}| - 1) \times l_{\text{gap}}$, where $|\nu|$ is the length of ν, $|\mathcal{V}|$ is the number of vehicles in $\mathcal{V}$, and l_{gap} is the car gap. Let $\mathcal{V}_1 \oplus \mathcal{V}_2$ be the vehicle sequence after appending $\mathcal{V}_2$ to the end of $\mathcal{V}_1$. Let $\mathcal{V}_1 \ominus \mathcal{V}_2$ be the vehicle sequence after removing all vehicles in $\mathcal{V}_2$ from $\mathcal{V}_1$. Let $\mathsf{head}(\mathcal{V})$ and $\mathsf{tail}(\mathcal{V})$ be the first vehicle and the last vehicle in $\mathcal{V}$, respectively. Let $\mathsf{rotate}(\mathcal{V}) = \mathcal{V} \ominus \langle \mathsf{head}(\mathcal{V}) \rangle \oplus \langle \mathsf{head}(\mathcal{V}) \rangle$ be the vehicle sequence after applying the rotation operation—moving the first vehicle of $\mathcal{V}$ to the end. Let $\mathsf{rotate}^k(\mathcal{V})$ be the vehicle sequence after applying the rotation operation k times to $\mathcal{V}$.

Algorithm 4 The 1-cyclability checking algorithm.

```
1: procedure is1Cyclable(ν, 𝒱, π)
2:   // ν is a new vehicle. 𝒱 is the vehicle sequence in ℛ(π).
3:   for i = 1 to |𝒱| do
4:     if not isMaximallyAssignable(𝒱 ⊕ ⟨ν, head(𝒱)⟩, π) then
5:       return False
6:     𝒱 := rotate(𝒱)
7:   return True
8: procedure isMaximallyAssignable(𝒱, π)
9:   Let ℛ(π) = ⟨R_1, R_2, ..., R_n⟩; i := 1; 𝒱_i := ⟨⟩
10:  for all ν ∈ 𝒱 do
11:    if ||𝒱_i ⊕ ⟨ν⟩|| ≤ ||R_i|| then
12:      𝒱_i := 𝒱_i ⊕ ⟨ν⟩
13:    else
14:      i := i + 1; 𝒱_i := ⟨⟩
15:      if i > n then return False
16:  return True
```

Algorithm 5 The greedily relocatability checking algorithm.

```
1: procedure isGreedilyRelocatable(ν, π)
2:   // ν is a new vehicle. ℛ(π) = {R_1, R_2, ..., R_n}
3:   Let 𝒱̄_i = 𝒱_i ⊖ ⟨head(𝒱_i)⟩ where 𝒱_i be the vehicles in R_i
4:   for i = 1 to |𝒱| do
5:     if not isGreedyRelocatableOfOneStack(𝒱̄_i) then
6:       return False
7:   return True
8: procedure isGreedilyRelocatableOfOneStack(𝒱̄_i)
9:   for i = 1 to n do 𝒱'_i := 𝒱_i
10:  for all ν ∈ reverse(𝒱̄_i) do     // 𝒱̄_i in the reverse order
11:    Find J = {j_1, ..., j_m} such that ||𝒱'_{j_k} ⊕ ⟨ν⟩|| ≤ ||R_{j_k}||
12:    if (J \ {i}) is empty then return False
13:    Find j ∈ (J \ {i}) such that ||𝒱'_j|| is minimum.
14:    𝒱'_j := 𝒱'_j ⊕ ⟨ν⟩
15:  return True
```

Definition 1 (Maximal Assignment): Given a sequence of queues $\mathcal{R}(\pi) = \langle R_1, R_2, \ldots, R_n\rangle$ managed by π, a *maximal assignment* of $\mathcal{V}$ to the queue group, denoted by $\mathsf{maximal}(\mathcal{V})$, is a partition $\langle \mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_n\rangle$ of $\mathcal{V}$, such that (1) $||\mathcal{V}_i|| \leq ||R_i||$, where $||R_i||$ is the length of R_i, for $1 \leq i \leq n$, and (2) there exist k such that $||\mathcal{V}_i \oplus \langle \mathsf{head}(\mathcal{V}_{i+1})\rangle|| > ||R_i||$ for $1 \leq i < k < n$ and $\mathcal{V}_j = \langle\rangle$ (i.e., an empty sequence) for $k < j \leq n$.

In other words, a maximal assignment puts as many vehicles as possible to the queues $R_1, R_2, \ldots, R_{k-1}$, and then put the remaining vehicles in R_k but leave $R_{k+1}, R_{k+2}, \ldots, R_n$ empty. A maximal assignment $\mathsf{maximal}(\mathcal{V})$ does *not* exist if k does not exist—i.e., after assigning as many vehicles as possible to $R_1, R_2, \ldots, R_n$, there are still some vehicles left that cannot be assigned to any queue.

Definition 2 (Cyclability): Given a sequence of queues $\mathcal{R}(\pi) = \langle R_1, R_2, \ldots, R_n\rangle$ in a queue group managed by π, a vehicle sequence $\mathcal{V}$ is *cyclable* in $\mathcal{R}(\pi)$ if and only if $\mathsf{maximal}(\mathsf{rotate}^i(\mathcal{V}))$ exists for all $0 \leq i < |\mathcal{V}|$.

If $\mathcal{V}$ is cyclable in $\mathcal{R}(\pi)$, it means that no matter how $\mathcal{V}$ rotates, it is still possible to partition $\mathcal{V}$ and assign each partition to each queue without exceeding the queues' capacity. Hence, if the sequence of all vehicles in $\mathcal{R}(\pi)$ is cyclable, there is no gridlock in the queue group.

A queue manager π can decide whether it should accept a request to park a vehicle ν in its queue group by checking whether the vehicle sequence in the queue group remains cyclable after adding ν. More precisely, suppose π receives a request to park ν in the queue group $\langle R_1, R_2, \ldots, R_n\rangle$ at time t. Without loss of generality, π plans to add ν to the end of R_n. Let $\mathcal{V}_i$ be the vehicle sequence in R_i at time t for $1 \leq i \leq n$. Let $\mathcal{V} = \bigoplus_{1\leq i\leq n} \mathcal{V}_i$ be the sequence of all vehicles in the queue group before adding ν to R_n. If $\mathcal{V} \oplus \langle\nu\rangle$ is cyclable, then π should accept the request.

However, there is a catch. Even if $\mathcal{V}$ is cyclable in a queue group, it may require *several* vehicles to move from one queue in $\mathcal{R}(\pi)$ to the next queue simultaneously to avoid gridlock. This simultaneous movement can potentially lead to gridlocks in the shared regions, as we shall discuss in Sec. IV-C. Therefore, we opt for a stronger notion of cyclability called *1-cyclability*, which only requires one vehicle to use a shared region at a time.

Definition 3 (1-Cyclability): Given a sequence of queues $\mathcal{R}(\pi)$, a vehicle sequence $\mathcal{V}$ is *1-cyclable* in $\mathcal{R}(\pi)$ if and only if $\mathsf{maximal}(\mathsf{rotate}^i(\mathcal{V}) \oplus \langle \mathsf{head}(\mathsf{rotate}^i(\mathcal{V}))\rangle)$ exists for $0 \leq i < |\mathcal{V}|$.

If $\mathcal{V} \oplus \langle \mathsf{head}(\mathcal{V})\rangle$ is cyclable in $\mathcal{R}(\pi)$, it means that the first vehicle in $\mathcal{V}$ can always move to the end of R_n without requiring some vehicles in R_n to move to R_{n-1}. Then there are enough room for some vehicles to move from R_2 to R_1, and then from R_3 to R_2, and so on. This rotation operation can proceed by moving one vehicle at a time. To avoid gridlock, π check whether $\mathcal{V} \oplus \langle\nu\rangle$ is 1-cyclable before accepting the request to add ν to the queue group. Algorithm 4 is the pseudocode of the algorithm for checking 1-cyclability. *A queue manager is 1-cyclable if it maintains 1-cyclability at all time.*

B. Greedily Relocatability of Stack Groups

A stack group does not suffer from gridlock if there is enough room in the adjacent shared regions to hold the vehicles that need to move temporarily. For example, in Fig. 4, if ν_{10} requests to leave the stack group, ν_{11} and ν_{12} can move to shared region temporarily and then return to R_5 after ν_{10} left. However, shared regions are shared resources, and they should not be used to hold vehicles temporarily. Therefore, we prefer relocating vehicles to other stacks instead. Unfortunately, ν_{11} and ν_{12} in Fig. 4 cannot relocate to other stacks due to the lack of space in the stack group. Therefore, we want to guarantee that at any moment, a stack group has enough space for relocation in the worst case, such that the situation in Fig. 4 would not occur.

Let us state the condition of this guarantee precisely. Let $\mathcal{R}(\pi) = \{R_1, R_2, \ldots, R_n\}$ be the regions managed by a stack manager π. Let $\mathcal{V}_i$ be the sequence of vehicles in stack R_i, for $1 \leq i \leq n$. Let $\mathsf{head}(\mathcal{V}_i)$ be the vehicle at the bottom of stack R_i if the stack is not empty. Let $\bar{\mathcal{V}}_i = \mathcal{V}_i \ominus \langle \mathsf{head}(\mathcal{V}_i)\rangle$ be the vehicle sequence above $\mathsf{head}(\mathcal{V}_i)$. $\bar{\mathcal{V}}_i$ is *relocatable* if there is a partition $\{\bar{\mathcal{V}}'_1, \ldots, \bar{\mathcal{V}}'_{i-1}, \bar{\mathcal{V}}'_{i+1}, \ldots, \bar{\mathcal{V}}'_n\}$ of the set of vehicles in $\bar{\mathcal{V}}_i$ such that $||\mathcal{V}_j \oplus \mathcal{V}'_j|| \leq ||R_j||$ for $1 \leq j \leq n$ and $j \neq i$.

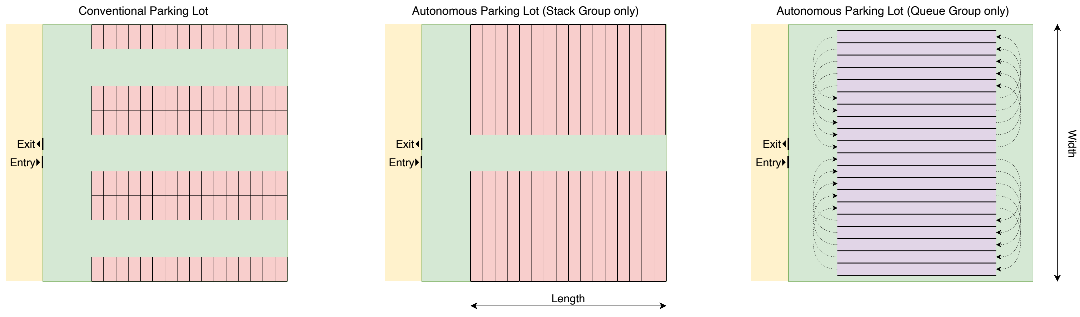

Fig. 5: The layout of the parking lots in the experiments.

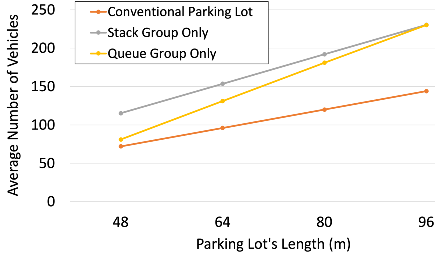

Fig. 6: The average number of vehicles in the parking lots versus the parking lot's length.

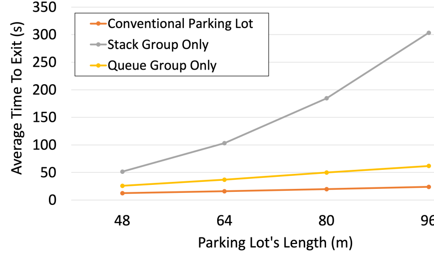

Fig. 7: The average time the vehicles took to exit a parking lot after it decides to leave, versus the parking lot's length.

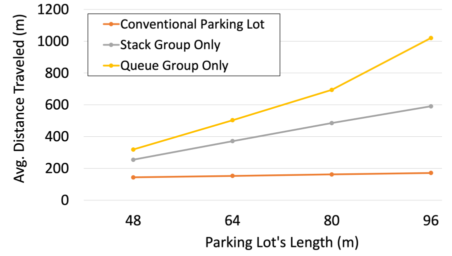

Fig. 8: The average distance vehicles traveled inside a parking lot, versus the parking lot's length.

Definition 4 (Relocatability): The vehicles in a stack group $\{R_1, R_2, \ldots, R_n\}$ is relocatable if and only if $\bar{\mathcal{V}}_i$ is relocatable for all $\mathcal{V}_i$, the vehicle sequence in R_i, $1 \leq i \leq n$.

However, it is difficult to check the condition in the above definition because the problem of finding a partition for $\bar{\mathcal{V}}_i$ subsumes the bin packing problem, a famous NP-hard problem. Therefore, we opt for a stronger version of relocatability called *greedy relocatability*, which always relocates a vehicle to a stack with the largest empty space. It is cumbersome to define greedily relocatability mathematically; instead, we present Algorithm 5 for checking greedily relocatability. A stack manager can use Algorithm 5 to check whether greedily relocatability can be maintained after adding a vehicle. *A stack manager is greedily relocatable if it maintains greedily relocatability at all times.*

C. Sufficient Conditions for Gridlock-free APLs

Gridlock-free is an important property for APLs because all vehicles in an APL should be able to leave the parking lot. Our previous work on AIM provides the sufficient condition under which an autonomous traffic network with multiple intersections will be live (i.e., gridlock-free) [14]. But the condition is inapplicable to APLs. The following theorem states a sufficient condition for gridlock-free APLs.

Theorem 1: If (1) all reservation handlers are fair, (2) all queue managers are 1-cyclable, and (3) all stack managers are greedily relocatable, the APL is gridlock-free.

Sketch of Proof. We assume (1) all vehicles in the parking lot will eventually request to leave the parking lot, and (2) the exits will not be blocked indefinitely. We say the RH of a shared region R is *fair* if all managers adjacent to R have a chance to use R if the managers repeatedly send requests to the RH of R—i.e., no request will be denied indefinitely. A RH of R that uses Eq. 1 to prioritize requests can guarantee that the RH is fair because the priority of denied requests will grow exponentially so that these requests will eventually have the highest priority. Since all vehicles in R will leave R, the RH will eventually be able to reserve tiles for these requests. Moreover, since all vehicles will eventually leave the parking lot successfully as no exit will be blocked forever, eventually there will be one managed region that can accept the vehicles or one exit that the vehicle can leave. Hence, all requests to a RH will eventually be accepted if the managers sent them repeatedly.

If all queue managers are 1-cyclable and all stack managers are greedily relocatable, there is no gridlock in the managed regions. More importantly, these managers can achieve gridlock-free by sending vehicles to shared regions one at a time. Hence, there are no two requests to the RHs that depend on each other. The lack of dependency of requests guarantees that the requests to the RHs will eventually be accepted due to the fairness of the RHs. Thus the managed regions will continue to be either 1-cyclable or greedily relocatable. In summary, all vehicles can eventually park in the parking lot and then leave the parking lot. Hence, the APL is gridlock-free. □

V. EXPERIMENTAL EVALUATION

We conducted a preliminary experiment to compare autonomous parking lots with conventional parking lots. Our hypothesis is that autonomous parking lots will be more space-efficient, but vehicles will take a longer time to leave the parking lots. Our experiments were based on a C++ simulator we developed. The GUI was implemented in PyGame 2.0 and connected to the simulator via gRPC. Fig. 5 showed the three parking lots we implemented in the simulator: a

conventional parking lot, an APL with stack groups only, and an APL with queue groups only. For a fair comparison, the dimensions of these parking lots are the same. The parking lot's width is 84 m, but the length of the parking lot varies from 48 m to 96 m, excluding the shared region on the left. Then we fit as many stack groups and queue groups as possible in the parking lot. There are one entry and one exit. The sizes of vehicles are random, but the maximum length is 4.8 m, and the maximum width is 2.4 m. The vehicles' kinematic model is based on the unicycle model, which offers simplified car-like vehicle dynamics. In this model, the set of differential equations for nonholonomic motion are:

$$\frac{\partial x}{\partial t} = v \cdot \cos(\phi), \quad \frac{\partial y}{\partial t} = v \cdot \sin(\phi), \quad \frac{\partial \phi}{\partial t} = v \cdot \frac{\tan \psi}{L}, \tag{2}$$

where (x, y) is the position of the center of the front of a vehicle, ϕ is the direction of the vehicle, and L is the length of the vehicle's wheelbase. The position and the direction depend on the steering angle ψ and the velocity v.

We set the vehicle's spawn rate at the entry to a large value such that vehicles will keep trying to enter the parking lot. Vehicles will randomly choose to leave the parking lot at any moment, but the probability of leaving is much lower than the spawn rate. Hence the parking lot will eventually be full. After that, we start to measure three performance metrics: (1) the average number of vehicles in a parking lot at any time step, (2) the average time a vehicle takes to exit a parking lot after it decides to leave, and (3) the average distance a vehicle travels after it enters a parking lot and before it leaves. We plotted these performance measures against the parking lots' length, as shown in Fig. 6-8.

Fig. 6 confirmed our hypothesis that autonomous parking lots are more space-efficient—about 60% more space-efficient than the conventional parking lot when the parking lot's length is $96\ m$. The queue groups were not as space-efficient as the stack groups when the parking lot's length is small, because the queue groups require more space for vehicles' U-turns. As the parking lot's length increases, the queue groups eventually became as space-efficient as the stack groups. Unsurprisingly, Fig. 7 and 8 show that the vehicles in the conventional parking lot took less time to exit the parking lot, and they do not move too much inside the parking lot. The vehicles in the stack groups take a much longer time to leave because congestion occurred at the middle driveway. There was no congestion in APLs with queue groups only, thanks to the AIM reservation system.

VI. CONCLUSIONS AND FUTURE WORK

Parking space is a scarce resource in urban environments. Previous works on HDP showed that we could dramatically increase parking lots' capacity by leveraging autonomous driving. However, most existing works have not considered mixing different parking strategies, ignored the gridlock issue, and assumed all vehicles occupy the same amount of parking space. This paper addressed these issues and presented a new gridlock-free autonomous parking lot design for HDP. The key ideas are (1) different regions in a parking lot are managed by different management agents, and (2) utilizing a reservation system for managing traffic between the regions. Our parking lot design can fit both stacks and queues, the two common parking schemes in HDP, into a parking lot. Our experiment confirmed a 60% increase in parking lots' capacity as suggested in some previous studies. However, we also showed that the debate about whether stacks are better than queues is far from over.

In the future, vehicles can be quite different from today's vehicles. Our parking lot design is feasible enough to accommodate other kinds of managed regions that accept robots with different motion constraints. Thus, our APLs could be used as a robot storage system for autonomous mobile robots. Moreover, we intend to figure out the best configuration of managed regions for a very large autonomous parking lot.

ACKNOWLEDGMENT

This work has been taken place at UNIST and was supported by NRF (2016R1D1A1B0101359816 and 2016M3C4A795263722) and UNIST (1.210080.01).

REFERENCES

[1] J. Timpner, S. Friedrichs, J. van Balen, and L. Wolf, "k-stacks: High-density valet parking for automated vehicles," in *IEEE Intelligent Vehicles Symposium*, 2015.

[2] H. Banzhaf, F. Quedenfeld, D. Nienhüser, S. Knoop, and J. M. Zöllner, "High density valet parking using k-deques in driveways," in *IEEE Intelligent Vehicles Symposium*, 2017, pp. 1413–1420.

[3] P. d'Orey, J. Azevedo, and M. Ferreira, "Automated planning and control for high-density parking lots," in *International Conference on Automated Planning and Scheduling (ICAPS)*, 2017.

[4] J. Azevedo, P. M. D'orey, and M. Ferreira, "High-density parking for automated vehicles: A complete evaluation of coordination mechanisms," *IEEE Access*, vol. 8, pp. 43 944–43 955, 2020.

[5] J. Hill, G. Rhodes, and S. Vollar, *Car Park Designers' Handbook*. ICE Publishing, 2013.

[6] M. Nourinejad, S. Bahrami, and M. J. Roorda, "Designing parking facilities for autonomous vehicles," *Transportation Research Part B: Methodological*, vol. 109, pp. 110–127, 2018.

[7] M. Ferreira, L. Damas, H. Conceicão, P. M. d'Orey, R. Fernandes, P. Steenkiste, and P. Gomes, "Self-automated parking lots for autonomous vehicles based on vehicular ad hoc networking," in *IEEE Intelligent Vehicles Symposium*, 2014, pp. 472–479.

[8] P. M. d'Orey, J. Azevedo, and M. Ferreira, "Exploring the solution space of self-automated parking lots: An empirical evaluation of vehicle control strategies," in *IEEE International Conference on Intelligent Transportation Systems (ITSC)*, 2016, pp. 1134–1140.

[9] K. Dresner and P. Stone, "A multiagent approach to autonomous intersection management," *Journal of Artificial Intelligence Research (JAIR)*, March 2008.

[10] S. Ma, H. Jiang, M. Han, J. Xie, and C. Li, "Research on automatic parking systems based on parking scene recognition," *IEEE Access*, vol. 5, pp. 21 901–21 917, 2017.

[11] H. Banzhaf, D. Nienhüser, S. Knoop, and J. M. Zöllner, "The future of parking: A survey on automated valet parking with an outlook on high density parking," in *IEEE Intelligent Vehicles Symposium*, 2017, pp. 1827–1834.

[12] P. Zips, H. Banzhaf, G. Quast, and A. Kugi, "Increasing the capacity for automated valet parking using variable spot width," in *IEEE International Conference on Intelligent Transportation Systems (ITSC)*, 2020, pp. 1–6.

[13] N. Shahidi, T.-C. Au, and P. Stone, "Batch reservations in autonomous intersection management," in *Proceedings of the International Joint Conferenceon Autonomous Agents and Multi Agent Systems (AAMAS)*, 2011.

[14] T.-C. Au, N. Shahidi, and P. Stone, "Enforcing liveness in autonomous traffic management," in *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 2011, pp. 1317–1322.