Scheduling of Mobile Workstations for Overlapping Production Time and Delivery Time

Dohee Lee[1] and Tsz-Chiu Au[1]

Abstract— **Many existing mobile service robots, including the robots in Robocup@Home, perform their designated tasks only when the robots are stationary. The efficiency of these robots can be improved if they can perform some tasks while moving. In this paper, we propose the concept of *mobile workstations*, which combine mobile platforms with production machinery to increase efficiency by overlapping production time and delivery time. We present a model of mobile workstations and their jobs and describe the task planning algorithm for a team of mobile workstations. The temporal planning problem for mobile workstations combines both features of job shop scheduling problems (JSP) and traveling salesman problem (TSP), but there is little work in the literature that tackles both JSP and TSP simultaneously. Our first algorithm is a complete search algorithm which returns an optimal temporal plan with minimum makespan, and our second algorithm conducts a local search in the space of task graphs so as to quickly return suboptimal temporal plans. According to our experiments, when the number of jobs is small, our second algorithm can generate near-optimal temporal plans, and when the number of jobs is large, our algorithm can generate much shorter plans than SGPlan 5 and a version of job shop scheduling algorithms.**

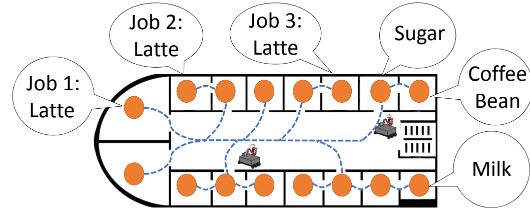

Fig. 1: A scenario for mobile coffee making robots.

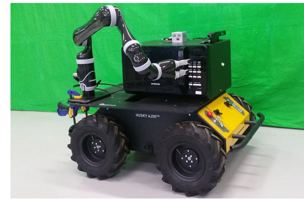

Fig. 2: A mobile microwave robot.

Fig. 3: A mobile printer robot.

I. Introduction

Mobility is essential in many tasks that service robots are designed to perform. Attaching a robot to a mobile platform is currently the most prevalent way to endow robots with mobility. Some mobile platforms allow users to attach any equipment to them easily. For example, a mobile platform called Patin by Flower Robotics[1] allows users to install household appliances such lamps and air cleaners on it. However, few research have considered attaching more sophisticated devices that can perform some production tasks *while* the mobile platform is moving. For examples, robots in Robocup@Home can get some tasks done only when the robots stop at locations with necessary appliances. Likewise, robots in Robocup@Work robots can only operate machines at stationary workstations. In this paper, we consider attaching production machinery to a mobile platform and study the implication in the scheduling of tasks for service robots. We call these robots *mobile workstations*, which can produce or process goods while delivering the goods. We argue that this mode of operation offers a new way to save time substantially. For example, consider the scenario in Fig. 1 in which you want to get a cup of coffee. If the coffee machine is located far from your office, you can save time by ordering the coffee machine to make a cup of coffee and then ask a robot to deliver it to you. However, you can save even more time if the robot equips with a coffee maker that can do two jobs simultaneously: brewing a cup of coffee and delivering it to your office. By overlapping production time and delivery time, mobile workstations can substantially save time and serve a larger number of customers.

Mobile workstations generalize the concept of mobile coffee making robots to any mobile robots with production machinery. For example, Fig. 2 and Fig. 3 show our hardware implementation of two different mobile workstations, one for microwave cooking and the other for printing. Our model of mobile workstations and our task planning algorithms are general enough to encompass all these machines. The success of these mobile workstations depends on careful timings of its production and its movement such that the production time and movement time are overlapped as much as possible. We consider the temporal planning problem of *multiple* mobile workstations: given N mobile workstations and M jobs, how can we assign the jobs and schedule the actions for the workstations such that the entire group of workstations can finish the jobs in the least amount of time? The minimization of the makespan makes sense especially when there are several mobile workstations that can work in parallel. Whenever a job is added to or removed from the job queue, our system will rerun the planning algorithm for replanning such that the optimality can be maintained.

Our planning problem is a NP-hard problem since it subsumes two difficult NP-hard problems: the job shop

[1]School of Electrical and Computer Engineering, Ulsan National Institute of Science and Technology, Ulsan, South Korea. {dohee,chiu}@unist.ac.kr

[1]http://www.flower-robotics.com/patin

scheduling problem (JSP) and the traveling salesman problem (TSP). The JSP and the TSP are famous problems and a large body of work for solving each of them has been accumulated. However, there are not many works in the existing planning and scheduling literature that focus exclusively on solving both JSP and TSP simultaneously within one framework. Therefore, we tackle this problem by devising a new algorithm which searches in the space of task graphs whose nodes corresponding to the production and movement actions. After finding the optimal task graph with the shortest critical path, our algorithm converts the task graph into a temporal plan. However, like many temporal planning problems, a complete algorithm for generating an optimal plan will run in exponential time (unless P = NP) and cannot practically solve any problems with a dozen of jobs. Hence, our second algorithm conducts a local search in the space of task graphs so as to quickly generate a suboptimal temporal plan for a large number of jobs. We conducted experiments to compare our algorithms with SGPlan 5, one of the best domain-independent temporal planners [1], [2] that uses Planning Domain Definition Language (PDDL) [3], which is expressive enough to define our problem. There is no domain dependent planner to solve our problem, but we also compared our algorithms with the JSP solver in Google Optimization Tools[2] using a set of test problems that the JSP solver can solve. Our experimental results show that our local search algorithm outperforms both SGPlan 5 and Google's JSP solver. More importantly, our algorithm can generate temporal plans whose makespans are not much worse than the optimal solutions when the problem size is small.

This paper is organized as follows. After presenting the related work, we describe our model of mobile workstations and their jobs. Then we define the multiagent planning problem and present an algorithm to solve the problem. After that we present our experimental results comparing our algorithm with the best temporal planner in the literature. Finally, we present the summary and the future work.

II. Related Work

Our idea is partially inspired by the recent advances in warehouse robots, notably the automated guided vehicles (AGVs) developed by Amazon Robotics (formerly Kiva Systems), that have revolutionized the operations of large distribution centers of online retailers. These warehouse robots lift storage pods in warehouses and move them to human workers for packing [4]. We think that further time saving can be attained if packing can start early when a pod begins to move. While this idea may not be realistic in warehouse settings due to various physical constraints, we explore the idea in other settings such as service robots. Scheduling algorithms are crucial in Kiva systems, which draws a grid in a warehouse and uses A* search to generate paths in the 2D grid while minimizing the travel time. In [5], Kiva systems reported the algorithms used for resource allocations and solving the corresponding global optimization problem.

[2] https://developers.google.com/optimization/scheduling/job_shop

In recent years there were several competitions on robotics in manufacturing settings. One noticeable event is Robocup@Work, which targets the use of robots in work-related scenarios [6]. In Robocup@Work, mobile manipulators deal with tasks ranging from manufacturing, automation, and parts handling up to general logistics (e.g., [7]). A more recent competition is the RoboCup Logistics League, which also focuses on in-factory logistics applications, with a stronger emphasis on planning. In this competition, robots fetch raw materials from input storages, transport them between stationary machines, operate the machines, and deliver the final products. The IEEE Virtual Manufacturing Automation Challenge (VMAC) is a competition about controlling multiple AGVs to transport various goods between several input and output locations in the warehouse environment [8]. Our concept of mobile workstations is partially inspired by the fact that machines in these competitions cannot move.

Subsumed within our task planning problem is the pickup and delivery problem (PDP), which has been studied in various settings [9]. As a subclass of vehicle routing problems, objects or people have to be picked up and delivered quickly [10]. Like the traveling salesman problems (TSPs), these problems are NP-hard, and there is no efficient algorithm to solve them optimally. There are heuristic solutions such as the one-to-one method [11] and Load LIFO [12]. [13] studied a generalized version of PDPs that includes various characteristics found in many practical PDPs. Clearly, our problem is even more complicated than General PDPs because we are solving two difficult problems—PDP and task planning—at once. The taxi dispatch problem is similar to PDPs, except that there are time limits about how long a customer can wait for a taxi [14], [15]. However, our work does not consider the deadlines for the task completion.

Our planning problem is related to the job shop scheduling problem (JSP), which is a famous optimization problem that concerns with the assignment of jobs to machines while minimizing the makespan. There are many efficient solvers for JSP (e.g., [16], [17]). The difference between JSP and our problem is that the mobility of machines, if any, is ignored in JSP. Although Beck, Prosser, and Selensky [18] proposed a way to formulate a vehicle routing problem (VRP) as a JSP, and vice versa, they did not attempt to combine both VRP and JSP into a single problem as we did. Another difference is that each job in JSP consists of the number of operations, which have to be processed in total order and each operation must be carried by specific machines [19]. In our problem, each subtask of a job can be assigned to different mobile workstations in a partial order. When compared with simultaneous task and motion planning (STAMP), our work focuses on task planning and path planning rather than motion planning.

III. Mobile Workstations and Their Jobs

We define the task planning model of mobile workstations as a directed tree that consists of N_{in} input nodes, N_{process} process node, and N_{out} output node, where $N_{\mathsf{in}} \geq 0$, $N_{\mathsf{process}} \geq 0$, and $N_{\mathsf{out}} > 0$. The input nodes represent the raw material

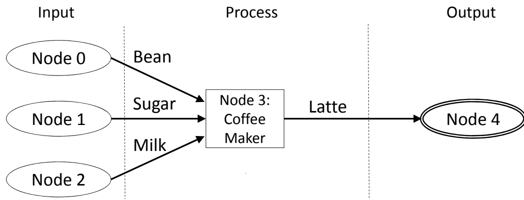

Fig. 4: A model of the mobile workstation for coffee making.

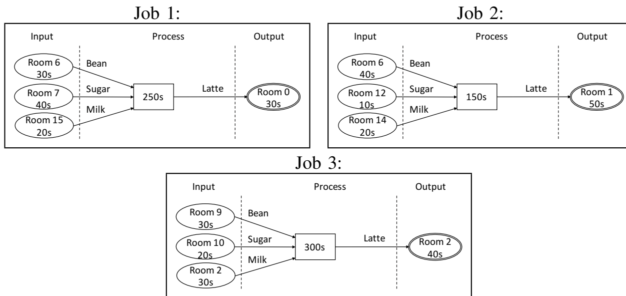

Fig. 5: Three jobs for the mobile coffee making robots

gathering devices such as robot arms, the process nodes represent the production machinery such as coffee makers and printers, and the output nodes represent the hardware for product delivery. As an example, a model of mobile coffee making robots is shown in Fig. 4. The incoming edges of the process node represent *prerequisites* for the production. It is possible to extend the model in Fig. 4 to have two or more process nodes as well as two or more output nodes, which means that the mobile coffee making robot is capable of producing two or more different products at the same time. To simplify our discussion, we focus on mobile workstations that have one process node and one output node only.

Clearly, different models of mobile workstations accept different types of jobs. The jobs for mobile printer robots are different from the jobs for mobile coffee making robots. Fortunately, all scheduling algorithms need to know is 1) the dependency of the subtasks in a job and 2) the timing of each subtask. Thus we can ignore the difference between different types of jobs and define a general model of jobs for mobile workstations. No matter what types of mobile workstations you are talking about, a job for a mobile workstation will first be translated into a graph similar to those in Fig. 5, which are jobs for the model of mobile coffee making robots in Fig. 4. In this paper, we call such graph a *job*. A job is a graph that is the same as the model of the corresponding mobile workstations, except that each node are augmented with information such as 1) where to fetch the raw materials, 2) where to deliver the product, and 3) the maximum duration of each task in the model. For instance, each job in Fig. 5 is just like the model in Fig. 4, except that 1) each node contains a number which is the maximum duration of the action; and 2) the input and output nodes contain the locations where the actions should take place. In Job 1 in Fig. 5, the robot needs to spend at most 30s at Room 6 to grab some coffee bean, 40s at Room 7 to grab some sugar, and 20s at Room 15 to grab some milk. Note that our model allows the raw materials to be located at different locations, though it does not need to be the case in the real world. After that, the robot has to spend at most 250s to make a cup of latte. Then it has to spend at most 30s at Room 0 to deliver the latte. As you can see, the maximum duration of the same nodes in different jobs can be different. For example, Job 3 may have requested a large cup of latte, hence the robot needs to spend more time to brew coffee. We assume a workstation cannot move when it is fetching or delivering an object.

IV. The Temporal Planning Problem

A server is dedicated to handle all job requests. Upon receiving a job, the server will store the job in a *job queue*. When the job queue is updated due to the addition or removal of jobs, a planning algorithm will be used to update the current temporal plans for all mobile workstations. Suppose there are N mobile workstations and M jobs in the job queue. The planning algorithm generates a temporal plan π, which consists of four kinds of *durative* actions:

- $\mathsf{Fetch}(o,l;d)$—get an object o at a location l;
- $\mathsf{Process}(o,o_1,o_2,\ldots,o_n;d)$—make an object o from the objects o_1, o_2, $\ldots$, o_n;
- $\mathsf{Deliver}(o,l;d)$—deliver an object o at a location l; and
- $\mathsf{Move}(l;d)$—move to location l.

In all of these actions, d is the maximum duration, which is the longest time a robot takes to perform the action.

A temporal plan π is a schedule of these actions. Formally, π is a list of pairs, each of them is (t,a) where t is the beginning time for the execution of the action a. Figure 6 shows a plan for the three jobs in Figure 5. Some parameters of the actions in this plan are omitted since they can be inferred from Figure 5. The makespan of this plan is 610s. This temporal plan is *optimal*, in the sense that there is no other temporal plan whose makespan is less than 610s.

V. Planning Algorithms

Given N workstations and M jobs, we compute a temporal plan that minimizes the makespan. A forward exhaustive search algorithm is not efficient enough to find a plan even for a small number of jobs. Hence, we devise a new graph search algorithm based on a *task graph* $G=(V,E)$, in which each vertex in V represents a task and each edge in E represents the temporal ordering of two tasks. An edge $(v_1,v_2)\in E$ means the task at v_1 must finish before the task at v_2. When there is no edge between two vertices, the tasks can be handled in parallel because there is no other indirect dependency between them. Figure 7 shows the task graph for the three jobs in Figure 5, whose output nodes are connected to a new vertex represented as the diamond shape. A *supertree* is a subtree denoted by the black arrows in Figure 7. A task graph can be constructed from a supertree by adding (1) exclusive-task edges and (2) movement vertices and movement edges. There are many ways to add these edges and vertices (see below), thus many different task graphs can be derived from a supertree. The *critical path* of a task graph is the longest path from the root to a vertex, where

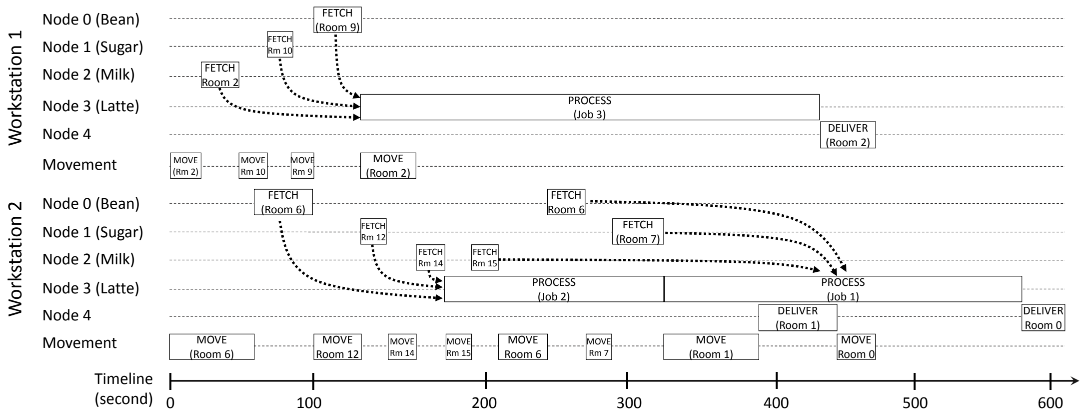

Fig. 6: An optimal temporal plan for the three jobs in Figure 5.

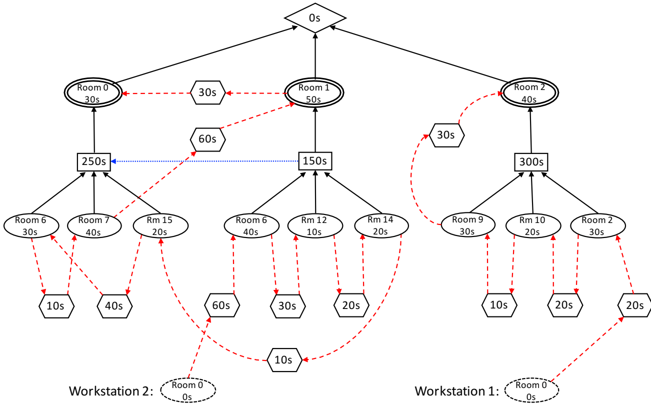

Fig. 7: A task graph for the jobs in Figure 5

the length of a path is the sum of the maximum duration of all vertices on the path. Then we employ the critical path method (CPM) for scheduling a set of project activities [20]: among all possible task graphs generated from a supertree, find one that has the shortest critical path. Our innovation is on how to quickly enumerate all possible task graphs.

Exclusive-Task Edges: Consider a supertree T formed by joining a set of M jobs: $J_1, J_2, \ldots, J_M$. We uniquely label the vertex for a node i in a job J_j by (i, j). First, we partition the set of jobs into N partitions: $\rho^0 = \{\mathbb{J}_1, \mathbb{J}_2, \ldots, \mathbb{J}_N\}$, such that all jobs in $\mathbb{J}_k$ will be assigned to the workstation k. In each partition $\mathbb{J}_k$, we add a set of edges to T to represent the constraints that a process node of the workstation k cannot perform two different jobs simultaneously. These mutually exclusive constraints can be encoded by a set of directed edges connecting the vertices (i, j) for every process node i and for every $J_j \in \mathbb{J}_k$. For each process node i, let $\rho^1_{(i,k)} = (j_1, j_2, \ldots j_{|\mathbb{J}_k|})$ be a *permutation* of the indices of the jobs in $\mathbb{J}_k$. Then we add an *exclusive-task* edge connecting (i, j_x) to (i, j_{x+1}), for $1 \le x < |\mathbb{J}_k|$. For example, in Figure 7, the blue line is an exclusive-task edge $((3,2),(3,1))$ that enforces the ordering that the vertex $(3,1)$ must be performed after $(3,2)$.

Movement Vertices and Movement Edges: We add a set of *movement vertices* and *movement edges* to T to represent the path of the workstation k. In T, all vertices for the input nodes and the output nodes in partition $\mathbb{J}_k$ are associated with the locations the workstation k should visit. Moreover, the workstation must stay at these locations for a given duration to fetch or deliver an object. Let $\{v_1, v_2, \ldots, v_r\}$ be the set of all vertices of the input nodes and output nodes of the jobs in $\mathbb{J}_k$, l_x be the location of v_x, for $1 \le x \le r$, and l_0 be the initial location of the workstation k. Given a permutation $\rho^2_k = (s_1, s_2, \ldots, s_r)$ of the first r positive integers, we add a new vertex $v^{\text{move}}_{(l_{s_i}, l_{s_{i+1}})}$ between v_{s_i} and $v_{s_{i+1}}$, for $1 \le s < r$, and vertex $v^{\text{move}}_{(l_0, l_{s_1})}$ and an edge $(v^{\text{move}}_{(l_0, l_{s_1})}, v_{s_1})$ as well as the edges $(v_{s_i}, v^{\text{move}}_{(l_{s_i}, l_{s_{i+1}})})$ and $(v^{\text{move}}_{(l_{s_i}, l_{s_{i+1}})}, v_{s_{i+1}})$ to T. These movement vertices and movement edges are the hexagons and the red edges in Figure 7, respectively. The maximum duration of a movement vertex $v^{\text{move}}_{(l_{s_i}, l_{s_{i+1}})}$ is the time the robot takes to move from l_{s_i} to $l_{s_{i+1}}$.

Our complete search algorithm computes the shortest critical path by enumerating all possible tuples of form $(\rho^0, \{(\rho^1_{(i,k)}, \rho^2_k)\}_{1 \le k \le N})$, which describes the set of all new vertices and edges in T. For each tuple, we use a depth-first search to compute the length of the critical path. Then we identify the tuple that gives the shortest critical path. Notice that ρ^2_k must be a *topological ordering* in order to maintain the temporal dependencies between the input or output nodes. The running time of this enumeration algorithm is $O(M^N \times M! \times r! \times (|V| + |E|))$, where N is the number of workstations, M is the number of jobs and $r = |V_{\text{IO}}|$ is the total number of input and output nodes in the supertree.

While this complete search algorithm guarantees to return an optimal solution, this is not a practical algorithm due to its exponential running time. In fact, like TSPs, our problem is NP-hard, meaning that any complete algorithm has little hope to return an optimal solution for any large problems in a reasonable time. Therefore, we devise a local search algorithm that can quickly solve large problems suboptimally. The pseudo-code of the algorithm is shown in Algorithm 1. Instead of exhaustively enumerating all possible job partitions, exclusive-task edges, movement vertices, and movement edges, the local search randomly selects a tuple $(\rho^0, \{(\rho^1_{(i,k)}, \rho^2_k)\}_{1 \le k \le N})$ and modifies it by randomly swapping the vertices in the permutations, with the hope that the solution can be improved (Lines 14 and

Algorithm 1 The local-search algorithm

1: **procedure** LOCALSEARCH($J_1, J_2, \ldots, J_m$)
2: Construct a supertree T from $J_1, J_2, \ldots, J_m$.
3: Let $D_{\min}$ be a very large number and let $S_{\min}$ be $\{\}$
4: **Repeat** the following N_{restart} times
5: Randomly generate a permutation ρ^0 of the jobs.
6: **Repeat** the following N_{repeat} times
7: Randomly generate permutations $\rho^1_{(i,k)}$ and ρ^2_k
8: $S := (\rho^0, \{(\rho^1_{(i,k)}, \rho^2_k)\}_{1 \le k \le N})$
9: Add exclusive-task edges to T according to $\rho^1_{(i,k)}$
10: Add movement vertices and edges to T w.r.t. ρ^2_k.
11: Compute length D of critical path in T
12: **If** $D < D_{\min}$ **then** $D_{\min} := D$; $S_{\min} := S$
13: Remove the exclusive-task edges and the movement vertices/edges from T
14: Randomly modify ρ^2_k by swapping without violating the topological order.
15: Randomly modify $\rho^1_{(i,k)}$ by swapping.
16: Add vertices and edges to T according to $S_{\min}$.
17: **return** the temporal plan π extracted from T.

15). These random swapping is similar to the min-conflict heuristics for local search that are used to solve the famous N-Queens problem [21]. Multiple sampling aims to improve the solution by repeating the inner loop N_{repeat} times with different swapping actions, whereas random restart reruns the local search from $N_{restart}$ different initial solutions. $S_{\min}$ stores the current best solution. In our experiments, both $N_{restart}$ and N_{repeat} are less than 5. The running time of the algorithm is $O(|V| + |E|)$, thus it can run very fast and can handle problems with many jobs. The algorithm can be transformed into an anytime algorithm by returning a temporal plan according to $S_{\min}$ at any time.

VI. Experimental Evaluation

We built two robots which work as mobile workstations, as shown in Fig. 2 and Fig. 3. We also implemented a web-based user interface for job submission as well as the monitoring of the mobile workstations. These robots work as intended in one of the floors of the building in our university. The implementation also deals with some practical issues such as replanning after a human user rescues a robot that gets stuck at a location. For more details, please take a look at the supplementary video.

The evaluation of our algorithms is mainly based on simulation since this allows us to test multiple robots working simultaneously in a much larger area. We compared our algorithms with SGPlan 5, one of the best temporal planners [1], [2]. In addition, we compared our algorithms with (1) a forward search algorithm that builds a temporal plan by exhaustively assigning tasks to the nodes in increasing time, and (2) a complete graph search algorithm that enumerates all possible task graphs as described in Section V. Due to the page limit we do not give a full account of these algorithms.

We wrote a simulator in Java and ran the experiments on a Linux cluster with 20 nodes and 120 cores equipped with Intel Core i7 CPUs running at 4GHz. We randomly generated

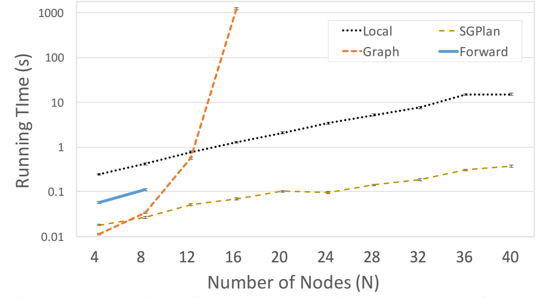

Fig. 8: Running times versus the number of nodes

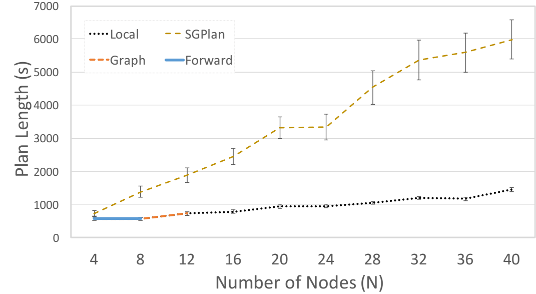

Fig. 9: Plan length versus the number of nodes

six different maps which model after the floor plans that are common in office and hotel environments. The time to move between two adjacent locations in a map is proportional to the distance between the locations plus a Gaussian error term which represents the variance of timing on different edges. We assume the robots are small enough such that they will not collide with each other and get stuck on the same road.

First of all, we conducted an experiment with one mobile workstation that includes up to 40 randomly generated nodes (including input, process, and output nodes). We randomly generated 120 problems with various numbers of jobs. The duration of a node is an integer randomly selected between $1s$ and $100s$. We grouped the problem instance according to the total number of nodes, and measured the running time and the plan length of the solution of the algorithms. The result is shown in Fig. 8 and Fig. 9. The error bars in these figures are the 95% confidence intervals. Each data point in the figures is an average of 120 values.

We set a time bound of 30 minutes such that if an algorithm cannot return a solution within the time bound, we declare that it fails to solve the problem instance. As shown in Fig. 8 and Fig. 9, the forward search algorithm and the graph search algorithm cannot solve any problem when the number of nodes is larger than 18, while the local search algorithm and SGPlan 5 can solve all problems within a few seconds. However, the quality of plans generated by our local search algorithm is much better than that of SGPlan (see Fig. 9). When the number of nodes is less than or equal to 14, the local search algorithm can generate plans whose length are almost the same the optimal plans generated by the forward search algorithm and the graph search algorithm. While we cannot tell how close the plans of the local search algorithm to the optimal plans when the number of nodes is larger than 14, we deduce the plans are nearly optimal, as considering the plan length grows slowly until the number

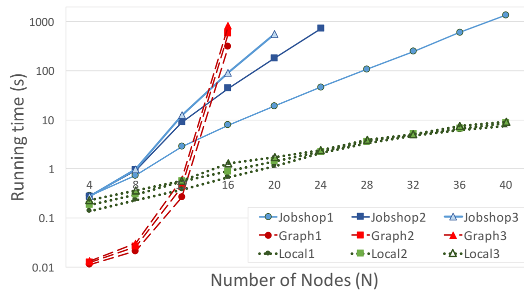

Fig. 10: Running times versus the number of nodes

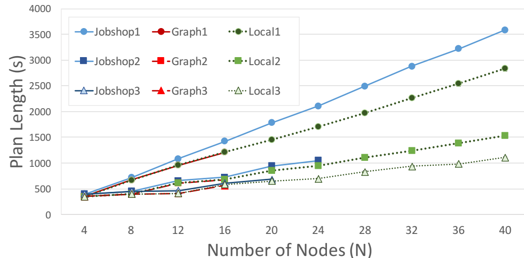

Fig. 11: Plan length versus the number of nodes

of nodes is 40 (see Fig. 9). But we do not know whether the solutions will be close to optimal for much larger problems.

Finally, we conducted an experiment to compare our approach with a job shop scheduling algorithm. However, there is no existing JSP solver that can handle movement actions like our algorithms do. For fair comparisons, we created a set of test problems in which all movement actions have the same duration if they move to the same location. This allows us to add the duration of movement to the duration of each task in a job such that a JSP solver can take the cost of movement into account without any additional effort. In addition, when two adjacent actions in a plan generated by a JSP solver occur the same location, we adjust the plan to remove the duplicated movement time. The settings of this experiment was the same as the one for SGPlan 5, except that we used the JSP solver in Google Optimization Tools instead. The results are shown in Fig. 10 and Fig. 11. When compared with Fig. 8 and Fig. 9, the JSP solver performed better than SGPlan 5 because the JSP solver does not need to solve the TSP problem embedded in our problem while SGPlan 5 did. However, the JSP solver did not perform as good as our local search algorithm in terms of both the running time and the plan length.

VII. Summary and Future Work

A mobile workstation combines a mobile platform with production machinery to increase efficiency by overlapping production time and delivery time. In this paper, we presented a complete, optimal, temporal planning algorithm for mobile workstations, and described a practical but suboptimal local-search algorithm that allows a small number of mobile workstations to handle a lot of jobs. Our experiments show that our local search algorithm outperforms Google's JSP solver as well as one of the best temporal planners called SGPlan 5 in terms of running time and solution quality. Currently, the algorithm plans for the maximum durations of actions and movements only in order to account for the variance of timing in low-level motion control. In the future, we intend to extend our algorithm to consider motion planning in order to optimize the path further by taking the precise timing of low-level control into account.

Acknowledgments

This work has been taken place in the ART Lab at Ulsan National Institute of Science & Technology. ART research is supported by NRF (2.190315.01 and 2.180557.01).

References

[1] Y. Chen, C.-W. Hsu, and B. W. Wah, "Sgplan: Subgoal partitioning and resolution in planning," *Edelkamp et al.(Edelkamp, Hoffmann, Littman, & Younes, 2004)*, 2004.

[2] Y. Chen, B. W. Wah, and C.-W. Hsu, "Temporal planning using subgoal partitioning and resolution in sgplan," *Journal of Artificial Intelligence Research (JAIR)*, vol. 26, pp. 323–369, 2006.

[3] M. Fox and D. Long, "PDDL2.1: An extension to PDDL for expressing temporal planning domains," *Journal of Artificial Intelligence Research (JAIR)*, vol. 20, pp. 61–124, 2003.

[4] P. R. Wurman, R. D'Andrea, and M. Mountz, "Coordinating hundreds of cooperative, autonomous vehicles in warehouses," *AI Magazine*, vol. 29, no. 1, p. 9, 2008.

[5] J. Enright and P. R. Wurman, "Optimization and coordinated autonomy in mobile fulfillment systems." in *Automated action planning for autonomous mobile robots*, 2011, pp. 33–38.

[6] G. Kraetzschmar, W. Nowak, N. Hochgeschwender, R. Bischoff, D. Kaczor, and F. Hegger, "Robocup@ work rulebook," 2013.

[7] T. Jenkel, "Mobile manipulation for the kuka youbot platform," Ph.D. dissertation, WORCESTER POLYTECHNIC INSTITUTE, 2013.

[8] D. Miklić, T. Petrović, M. Čorić, Z. Pišković, and S. Bogdan, "A modular control system for warehouse automation-algorithms and simulations in usarsim," in *Robotics and Automation (ICRA), 2012 IEEE International Conference on*. IEEE, 2012, pp. 3449–3454.

[9] G. Berbeglia, J.-F. Cordeau, I. Gribkovskaia, and G. Laporte, "Static pickup and delivery problems: a classification scheme and survey," *Top*, vol. 15, no. 1, pp. 1–31, 2007.

[10] G. Berbeglia, J.-F. Cordeau, and G. Laporte, "Dynamic pickup and delivery problems," *European journal of operational research*, vol. 202, no. 1, pp. 8–15, 2010.

[11] K. Treleaven, M. Pavone, and E. Frazzoli, "Asymptotically optimal algorithms for one-to-one pickup and delivery problems with applications to transportation systems," *Automatic Control, IEEE Transactions on*, vol. 58, no. 9, pp. 2261–2276, 2013.

[12] M. Cherkesly, G. Desaulniers, and G. Laporte, "A population-based metaheuristic for the pickup and delivery problem with time windows and lifo loading," *Computers & Operations Research*, vol. 62, pp. 23–35, 2015.

[13] M. W. Savelsbergh and M. Sol, "The general pickup and delivery problem," *Transportation science*, vol. 29, no. 1, pp. 17–29, 1995.

[14] W. Hao, "Improving taxi dispatch services with real-time traffic and customer information," Ph.D. dissertation, Ph.D. dissertation, National University of Singapore, 2004.

[15] K. T. Seow, N. H. Dang, and D.-H. Lee, "A collaborative multiagent taxi-dispatch system," *Automation Science and Engineering, IEEE Transactions on*, vol. 7, no. 3, pp. 607–616, 2010.

[16] A. P. E. Coffman Jr and R. L. Graham, "Optimal scheduling for two-processor systems," *Acta informatica*, vol. 1, no. 3, pp. 200–213, 1972.

[17] W. Xia and Z. Wu, "An effective hybrid optimization approach for multi-objective flexible job-shop scheduling problems," *Computers & Industrial Engineering*, vol. 48, no. 2, pp. 409–425, 2005.

[18] J. C. Beck, P. Prosser, and E. Selensky, "Vehicle routing and job shop scheduling: What's the difference?" in *International Conference on Automated Planning and Scheduling (ICAPS)*, 2003.

[19] P. Brucker and R. Schlie, "Job-shop scheduling with multi-purpose machines," *Computing*, vol. 45, no. 4, pp. 369–375, 1990.

[20] J. E. Kelley and M. R. Walker, "Critical-path planning and scheduling," in *Proceedings of The Eastern Joint Computer Conference*, 1959.

[21] R. Sosič and J. Gu, "Efficient local search with conflict minimization: A case study of the *n*-queens problem," *Knowledge and Data Engineering*, vol. 6, no. 5, pp. 661–668, 1994.