Alphabet Soup: A Testbed for Studying Resource Allocation in Multi-vehicle Systems

Christopher J. Hazard
cjhazard@ncsu.edu
North Carolina State University
Raleigh, NC 27695

Peter R. Wurman
wurman@ncsu.edu
North Carolina State University
Raleigh, NC 27695

Raffaello D'Andrea
rd28@cornell.edu
Cornell University
Ithaca, NY 14853

Abstract

We present ALPHABET SOUP, a Java-based model of a multi-vehicle warehouse that frames control and coordination issues. By presenting this abstract model of an actual system, we hope to expose the research community to the commercially consequential issues of resource allocation and robot motion planning. In ALPHABET SOUP, robots must be used to move buckets of letters from letter receiving stations to word-assembly stations. We discuss potential research problems, and in particular how the resource management problems are particularly well suited for auction-based resource management.

Introduction

The energy directed towards research on autonomous agents and multi-agent systems is fueled by the expectation that, in the near future, environments will be populated with hundreds or thousands of autonomous agents. The multi-agent programming paradigm has been shown to be an effective way to build and control complex systems (Jennings & Bussmann 2003). Combined with recent advances in robotic components, this approach makes it feasible to build large, complex systems of autonomous vehicles. Although systems with as many as 100 robots have been demonstrated, like the experimental CentiBot project (Konolige *et al.* 2004), the applications—disaster recovery or terrorist events—are not daily occurrences. Real, everyday applications with more than a few vehicles have been lacking.

Recently, the authors[1] have been involved with a company called Kiva Systems that is building low cost robots for pick-pack-and-ship warehouses. The key innovation in the Kiva system is the combination of inexpensive robots capable of lifting and carrying shelving units to and from pick stations. Workers stay at the stations, pick items off the shelves the robots present, and put the items into shipping cartons. By moving the inventory to the worker, rather than the other way around, the Kiva system provides a dramatic increase in worker productivity over competing approaches. The approach is also well suited for manufacturing or assembly operations. One thing that makes the Kiva system interesting to the research community is its size: a typical installation of a Kiva system in a large warehouse will involve several hundred robots and tens of thousands of movable shelving units.

Many engineering and computational challenges are associated with bringing a reliable, cost effective, *massively multi-vehicle system* (MMVS) to market. There is also the potential to apply various techniques developed by the research community to the problem domain. However, although there has been much research on the topics of multi-agent coordination, a great deal of it has been presented in the context of contrived problems. We believe the field can benefit from the availability of detailed yet high-level simulation environments that capture and focus on key elements of real multi-vehicle applications. By decoupling low-level physical and positional robot problems, which can be minimized in an aptly engineered and controlled warehouse environment, we can focus on the high-level algorithms.

Thus, we developed an abstraction of an MMVS approach to pick-pack-and-ship warehouses. We call it ALPHABET SOUP because the underlying task involves moving buckets of letters around a warehouse in order to assemble words. We have developed a Java-based simulation of ALPHABET SOUP[2] that is designed to provide a platform on which to study some of the key research questions entailed by a real MVS. The platform is designed to support two key research areas: 1) the coordination of multi-vehicle systems, and 2) resource allocation. This paper focuses more on the resource allocation problems entailed in the platform. Among the rich research resource allocation questions that can be studied in ALPHABET SOUP are:

- Where to store the buckets in the warehouse
- Which buckets to bring to which stations
- Which buckets to store new letters
- Which stations to assign words to
- Which stations to assign incoming letters

In the rest of the paper we present ALPHABET SOUP and the details of the testbed. We then discuss the above research questions in greater depth.

[1] The second and third author on sabbaticals, and the first author as a summer intern.

[2] Available at research.csc.ncsu.edu/alphabetsoup

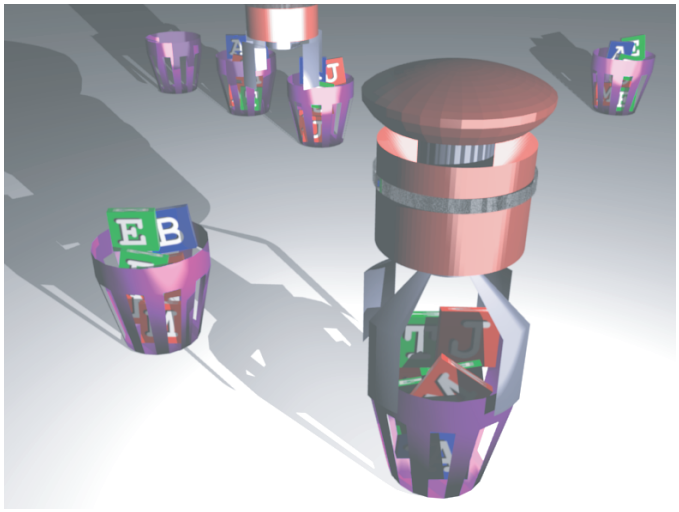

Figure 1: Conceptual drawing of ALPHABET SOUP

The Alphabet Soup Testbed

ALPHABET SOUP is analogous to the real-world problem of order fulfillment in a warehouse environment, or assembly in a manufacturing environment. The objective of the ALPHABET SOUP warehouse is to assemble specific words out of component letters. The inventory of the system are the *letter tiles*, which are stored in moveable buckets with fixed capacity. The buckets can be picked up and driven around the warehouse by *bucketbots*. The bucketbots are used to move buckets to and from stations to accomplish the overall system objectives. The *letter station* is used to put letter tiles into buckets, while the *word station* is used to take letters out of buckets and compose words. Stations can interact with the letter tiles in a bucket when the bucketbot has centered the bucket on the location of the station (within a tolerance). Stations are typically located on the borders of the map.

A *letter tile* is a combination of an English letter and a tile color, and a *word* is a sequence of letter tiles. The letters in a word do not need to have the same tile color. The testbed takes a word file–any text file of English words will do–and a color profile, and constructs a set of words. These words can then be distributed to the word stations as jobs that have to be completed. Each word station has a finite number of jobs it may be actively working on at any one time. Note that a station cannot take a letter out of a bucket that is not required for any of its active words. The act of taking a letter tile out of a bucket and putting it into position in a word takes a fixed amount of time. When a word is completed, the station puts it into the completed list and can accept a new word. The policy that is used to assign word jobs to stations is one area that can be studied in ALPHABET SOUP.

In order to build words, there must be an adequate inventory of letter tiles. New letters are received at the letter stations in homogeneous bundles of a fixed size. To get the letter tiles into inventory, one or more bucketbots must bring one or more buckets to the letter station. Obviously, the bucket must have enough free capacity to accept the number of letters the station attempts to store in it. Like the limit on the number of active words in a word station, each letter stations has a limit on the maximum number of bundles which may be simultaneously staged. The act of putting a letter into a bucket takes a fixed amount of time. The policy to assign letter bundles to letter stations, and to select which buckets in which to store the letters, are also areas that can be studied in ALPHABET SOUP.

In order to start a simulation run with enough inventory to immediately build words, the testbed includes an option to seed the buckets with letters. The initial inventory level is set as a fraction of the total warehouse capacity, and the profile of letter tiles in the buckets is drawn from the distribution of letters in the word file and colors in the color profile.

The final component of the system is the bucketbots, as conceptually illustrated in Figure 1. Each bucketbot has limited capabilities; it can grab a bucket, release a bucket, accelerate, decelerate, and tell a station to take a letter from, or put a letter into, the bucket it is carrying. A robot can pick up only one bucket at a time, and likewise a bucket may be attached only to one robot at a time. Robots may pass over/under buckets freely when they are not carrying another bucket. However, robots should not collide with other robots, and buckets should not collide with other buckets. When a collision occurs, all robots involved are completely stopped and penalized.

Figure 2 depicts the ALPHABET SOUP user interface. In the center of the figure is the graphic representation of the map, containing the letter stations on the left, word stations on the right, both as shaded circles. Bucketbots are shown as circles with lines indicating their orientation, and buckets are depicted as thicker, empty circles. Bucketbots which are straying from their desired path to evade a collision with another bucketbot—or another bucket if they carrying a bucket—are rendered with a thicker outline.

The mouse can be used to inspect the objects on the screen by selecting them. The left column of Figure 2 shows the contents of a selected letter station and a selected bucket. The selected letter station is highlighted on the center of the left side of the map, and the selected bucket is highlighted near the middle on the right side. The right column shows the list of completed words on the top, the open words in the selected word station in the middle, and the next words in the open word list in the bottom.

By releasing ALPHABET SOUP, we hope to make it easy for researchers to study algorithms and techniques that maximize sustainable word completion rate while minimizing the number of bucketbots, stations, and the total distance traveled.

Simulation Parameters and Metrics

ALPHABET SOUP has a number of configurable parameters to create a wide variety of problem scenarios. We expect that researchers focused on different subproblems will choose different combinations of parameters.

A warehouse has a configurable number of buckets, bucketbots, letter and word stations, all of which affect throughput. Additionally, the capacity of buckets and the size of letter bundles (placed into buckets by letter stations) are

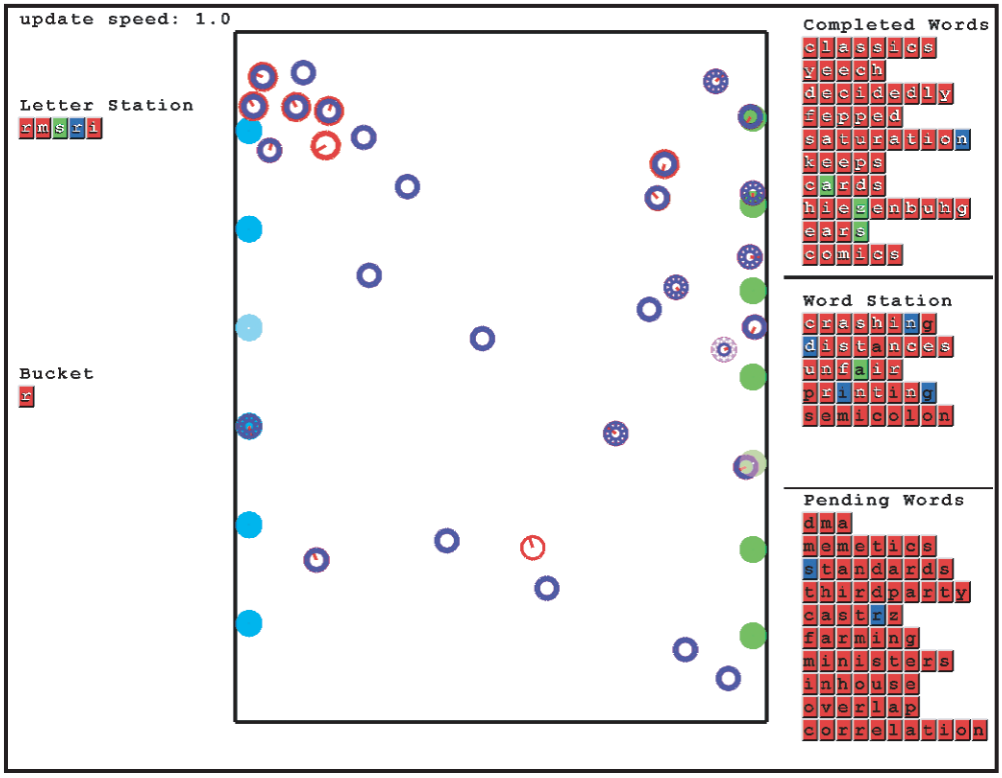

Figure 2: Screenshot of ALPHABET SOUP Testbed

also configurable. The choice of word dictionary and letter color distribution affects the number and profile of letters that must be stored as inventory in the warehouse. For instance, with a uniform distribution of colors and letters, each letter in every bucket is equally likely to be used. However, most sets of words will make more use of some letters (e.g., the letter 'e') than others (e.g. the letter 'z'). Further, a non-uniform color distribution will create even more variety in the frequency with which certain letter-color combinations are required. A profile with five hundred colors in a Pareto distribution would create 13,000 different letter tiles—on the order of the number of unique products in a large warehouse—with a letter tile profile something like the classic 80/20 curve.

The variations create some interesting opportunities. For instance, when using English words, buckets with the letter 'q' would benefit from also from having the letter 'u'. These associations between letters is analogous to associations between products which are frequently ordered together in a warehouse, such as cameras and camera cases. Additionally, one may want to store the popular colors together, and the unpopular colors together, so that more than one letter can be picked out of a bucket during most station visits.

The size of the physical objects, namely the warehouse, bucketbots and buckets, can be set in the configuration file. The latter directly affects how many are needed to store the inventory. These relative sizes affect bottlenecks of the system. Larger bucketbots and buckets, relative to the map size, restrict the available space to maneuver. With less available space, path planning, congestion avoidance, and spatial resource allocation are emphasized. On the other hand, smaller bucketbots and buckets emphasize bucketbot, bucket, and letter allocation strategies. Similarly, the configuration file also specifies how close a bucketbot must be to a bucket to pick it up, and how close it must be to a station to be considered present.

With regard to the numbers and capacities of physical objects, ALPHABET SOUP exhibits some basic relations. To achieve steady-state behavior utilizing available capacity, the throughput of the set of word stations should be balanced to the throughput of the letter stations. The number of bucketbots should be great enough such that stations do not sit idle, but also small enough such that bucketbot idle time is kept low and bucketbots are not continuously getting in the way of each other. The optimal number of buckets is obviously dependent on the size of the warehouse. For a large number of letter tile colors, more buckets are needed to make sure all letter tiles are represented, such that the letter stations do not become the bottleneck. As the bundle size increases, more total bucket capacity is needed to ensure that the system does not run out of storage space for new letters entering the system.

The temporal costs of various actions are also configurable. Key temporal actions include the amount of time that a bucketbot requires to pick up or set down a bucket, the amount of time that it takes to remove a letter tile, or

add one, to a bucket, and the amount of time it takes to move a finished word out of a word station and prepare for the next word. Bucketbot motion is described by its velocity and acceleration, both of which are configurable. These parameters, in turn, affect the bucket allocations for tasks, bucket storage, and letter placement strategies. The temporal penalty for bucketbot collisions is also configurable.

The testbed can run with or without a graphic display. Enabling graphics helps a developer visually test and debug algorithms, as well as gain intuition as to how algorithms are behaving. For running batch simulations, disabling the graphics reduces the overhead of real-time rendering and allows the testbed to run on remote terminals without requiring graphic support.

To easily support extensions, ALPHABET SOUP loads modules specified in its configuration file at runtime. These modules, which must inherit core classes and interfaces, allow the ALPHABET SOUP researcher to supply advanced behavior without modifying or needing to recompile any of the core modules.

To determine the effectiveness of a technique, ALPHABET SOUP tracks and reports of a number of statistics, including: the number of words completed, total number of letters in words completed, number of letters dispensed by letter stations, total and average distances driven by bucketbots, number of bucket grabs and releases, number of bucketbot and bucket collisions, bucketbot idle time, average bucket capacity utilization, average number of letter transfers per word/letter station visit, and station idle time.

Depending on the policies being studied, various components may become the bottleneck. If buckets can be delivered faster than stations can add or remove letters, then the maximum throughput is a function of the add/remove time and the number of letters per word. In such a case, the system is evaluated by how effectively it uses its bucketbots. However, if there are not enough bucketbots, they may not be able to deliver enough buckets to the stations to keep them busy. In that case, the throughput is the metric that measures overall system performance.

A potentially realistic scenario can be expressed with the following example parameters. Using the units of distance to mean meters and time to mean seconds, our modest-sized example warehouse is 250 meters by 350 meters. Bucketbots and buckets are each 2 meters in diameter. Bucketbots can accelerate at 20m/s^2 up to a maximum speed of 4m/s. This example warehouse contains 25 word stations, 25 letter stations, 250 bucketbots, and 850 buckets. With a bucket capacity of 40 letters, bundle size of 4, station time to move letters at 5.0 seconds, and bucket grab/release time at $\frac{1}{2}$ second, 4 colors with a distribution of $(\frac{4}{5}, \frac{1}{10}, \frac{1}{20}, \frac{1}{20})$, a dictionary of jargon with an average of 9.2 letters per word, and the example minimal coordination, we see throughputs of around one word per 20 seconds (based on elapsed time within the simulation). The minimal coordination simply assigns tasks first in, first out, requires buckets to be returned to storage between every task, and each task only involves one letter at a time. Bucketbot congestion and non-optimal task allocations are very obvious when watching the simulation. Based on observations and our experience in an industrial setting, coordination algorithms should be able to offer at least one to two orders of magnitude of improvement.

Bucketbot Movement

In the idealized ALPHABET SOUP environment, bucketbots have perfect traction, meaning that they cannot skid or slide. Besides collisions, the only movement constraints bucketbots have are maximum speed, V, and maximum acceleration, A. Given the bucketbot position (x, y), these constraints may be represented as,

$$\dot{x}^2 + \dot{y}^2 \leq V^2, \text{ and} \quad (1)$$

$$\ddot{x}^2 + \ddot{y}^2 \leq A^2. \quad (2)$$

To control bucketbot motion, bucketbot controls set a target velocity. The target velocity is comprised of components v_x and v_y. If the magnitude of the target velocity exceeds the maximum speed via equation 1, the target velocity vector is normalized to the maximum speed. Once this normalization has been performed, the acceleration constraint (equation 2) must be checked. As ALPHABET SOUP uses discrete time intervals, we will denote the time between updates as t. Given the current velocity, $(\dot{x}_0, \dot{y}_0)$, we can find the acceleration constrained velocity after the time interval, $(\dot{x}_t, \dot{y}_t)$, by first finding the actual magnitude of acceleration undertaken, a_t, to be

$$a_t = \sqrt{\left(\frac{v_x - \dot{x}_0}{t}\right)^2 + \left(\frac{v_y - \dot{y}_0}{t}\right)^2}. \quad (3)$$

If this magnitude of acceleration, a_t, does not exceed the maximal acceleration, A, then (v_x, v_y) will be used as the velocity of this timestep. However, if $a_t > A$, then the velocity of the this time step should be constrained to the maximal acceleration as

$$\dot{x}_t = \dot{x}_0 + \frac{A}{a_t}(v_x - \dot{x}_0), \text{ and} \quad (4)$$

$$\dot{y}_t = \dot{y}_0 + \frac{A}{a_t}(v_y - \dot{y}_0). \quad (5)$$

To minimize the simulation time required, the testbed only recomputes new positions and state transitions when an event occurs that could alter a bucketbot's acceleration or direction, or change the state of a letter, bucket, or station. The testbed is thus able to skip uneventful times of the simulation. The time to the next event is taken as the minimum possible time to the next event. To avoid situations similar to Zeno's Paradox[3], the time to next event is clamped with a lower bound of the time it would take any bucketbot to move the distance of its radius. Time until the next event is the minimum amount of time for any bucketbot to potentially collide, finish accelerating or decelerating, complete a

[3] If two bucketbots are about to collide, but continually change their directions and accelerations such that they will collide at a marginally later time, the next event will be a very short amount of time later. These increasingly small intervals of time prior to a collision increase the simulation time dramatically. With a minimum time to next event, the worst case is still reasonable.

turn, grab or release a bucket, finish transferring a letter, finish a specified amount of cruising time, or get close enough to another object such that the bucketbot may wish to change its plans.

When two or more bucketbots collide, their velocities and accelerations are immediately set to 0, and are given a timeout penalty. While the testbed could be extended to simulate elastic or inelastic collisions, we feel it is reasonable to assume, based on the coordinated and engineered environment, that bucketbots should not normally collide. Thus, we model collisions as extremely costly, negative events.

Bucketbot Sensing and Control

In ALPHABET SOUP a bucketbot potentially has perfect sensing capabilities; it can obtain all information about all other bucketbots and buckets within a specified distance. These sensing capabilities are due to the nature of the environment. Bucketbots can communicate with any other entity, and the entire system is engineered to maximize available information and precision. In many foreseeable practical applications of ALPHABET SOUP , the system is a controlled warehouse environment. To aid in sensing precision and information sharing, environments may be built with features such as wireless communication facilities, specially designed markings in the environment, and indoor positioning systems. Additionally, when any component exhibits an error or failure, the system can be paused for repair.

When the bucketbot "sees" another object, it can retrieve any information the system has about it, including direction, velocity, and bucket contents. The bucketbot also has full information about any bucket, bucketbot, or station that a separate managing process may provide. The bucketbot also knows its exact location, direction, and velocity. While bucketbots in ALPHABET SOUP are error-free, perfect sensing, and locally omniscient, these capabilities may be constrained for experimental applications by disregarding certain information.

A bucketbot must determine the length of time to accelerate and decelerate in to arrive at a specified location. Perhaps the simplest movement paradigm is for the bucketbot to stop between direction changes, maximally accelerating and decelerating when changing velocities. Our example minimalistic model uses this logic and only turns while in transit when evading another bucketbot or bucket. Because the bucketbot must decelerate back to a speed of 0 after moving, the speed reached during the acceleration phase must equal the speed at which the bucketbot can decelerate back to the speed of 0 during the deceleration time. From this, we can find the total acceleration time, t_{accel}, in terms of the maximum acceleration, A, distance to the goal, g, and initial velocity, v_0. For simplicity, t_{accel} need only be calculated on the axis with maximal acceleration as

$$t_{accel} = \frac{\sqrt{2}}{2a}\sqrt{2ag + v_0^2 - \sqrt{2}v_0}. \quad (6)$$

If the bucketbot will reach maximum velocity en route, it will need to cruise before beginning its deceleration. This maximum-velocity cruise time may be easily found after subtracting the acceleration and deceleration distances from the distance to the goal.

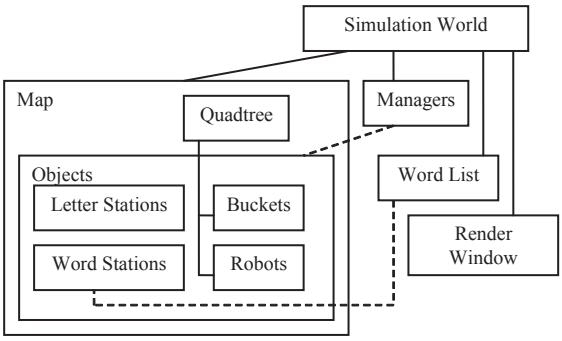

Figure 3: Architecture of ALPHABET SOUP Testbed

Architecture

ALPHABET SOUP has been designed to be easily extendable and useable by a wide audience. We chose Java and LWJGL[4] because they meet the following criteria: easy to build and run on most major platforms, fast execution and rendering, and the have wide acceptance and strong communities. The ALPHABET SOUP testbed itself is released under the GPL.[5]

To allow ALPHABET SOUP to run in batch mode and on machines without graphical rendering (such as many supercomputers), we have implemented a way to run the testbed in a "headless" mode. When running in headless mode, none of the classes that utilize the LWJGL library are loaded. The classes that perform rendering inherit from the base classes that perform the actual ALPHABET SOUP simulation. This inheritance scheme not only allows the rendering classes to display information based on the classes they extend, but also allows the rendering functionality to be distinctly separate from the simulation functionality.

Alphabet Soup Architecture

The basic ALPHABET SOUP Testbed architecture is summarized in Figure 3. SimulationWorld contains and constructs the rest of the framework. If ALPHABET SOUP is run with a graphic display, SimulationWorld loads RenderWindow and also loads all of the corresponding renderable classes for every object. SimulationWorld constructs everything according to the configuration parameters.

The map functions as a container for all of the physical objects and manages their interactions. The bucketbots and buckets are stored in a quadtree to optimize simulation performance. Quadtrees are a method of recursively dividing a space into regions based on the number of objects in each region. Our implementation uses a point-region quadtree; when the number of objects in a region exceed a maximum threshold, it divides the region into four equal areas with two

[4]Lightweight Java Game Library: www.lwjgl.org

[5]GNU General Public License: www.gnu.org/copyleft/gpl.html

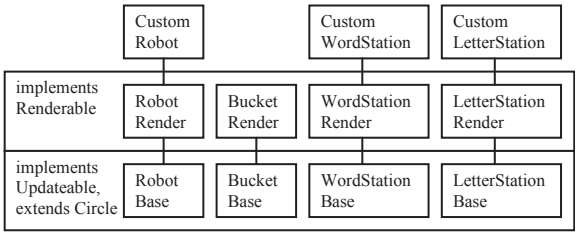

Figure 4: An example of extending ALPHABET SOUP.

cuts, and remerges subdivided regions when a minimal number of objects is reached. The quadtree greatly reduces algorithmic complexity of both detecting collisions and reporting bucketbots and buckets within a vicinity. To make sure adjacent regions are not discounted when searching for potential collisions or viewable objects, regions are expanded such that they have sufficient overlap.

The three major managers in the example ALPHABET SOUP controller implementation are the word manager, letter manager, and bucketbot manager. While the framework does not impose this manager architecture on implementations, we feel that this is a sensible approach. The word manager takes care of allocating words to stations, and communicates with the bucketbot manager about the allocations. The letter manager is similar to the word manager in that it controls which letters the letter stations produce, as well as communicates letter allocations to the bucketbot manager. The bucketbot manager coordinates all of the bucketbots, by manufacturing, prioritizing, and assigning tasks to bucketbots, buckets, letter stations, and word stations. In our default testbed, the bucketbots only keep track of one task at a time, and all planning other than avoiding obstacles and navigating to destinations is done in the bucketbot manager.

All of the managers can communicate with each other and also with the bucketbots, word stations, and letter stations using defined and extendable interfaces. If an object performs an action, other entities must ask the object to perform the action, rather than make the object perform the action itself. ALPHABET SOUP comes with some default example managers, which are intended to be extended or replaced. In terms of execution, all of the managers and objects have methods that are called when either their environment has changed or their timers have expired.

Extendable Interfaces

While any component of ALPHABET SOUP may be extended or modified, those best suited for studying control and allocation algorithms are the bucketbot behavior, word station policy, letter station policy, bucketbot manager, word manager, and letter manager. These particular entities may be changed by simply changing the configuration file.

Each of the physical objects held in the map extend a class called Circle which implements basic location and collision functionality. The object base classes also implement an interface named Updateable, which allows them to operate in the event driven model. Figure 4 illustrates this relationship, how the objects are extended to render themselves to the screen, and also one way a user of the testbed could extend these objects. The base functionality can be replaced or extended. Likewise, users may also override the way objects are rendered, or even leave out the rendering altogether.

With regard to resource management, such as bucket and letter selection and bucketbot coordination, the managers are the primary entities to modify. Several implementation schemes are possible. The bucketbot manager, letter manager, and word manager could each share equally prominent roles. A different solution would be to have one manager, such as the bucketbot manager, contain the majority of the logic and drive the other two lighter-weight managers. A further alternative would be to have all managers employ minimal logic and only function to keep track of resource utilization, while using the bucketbots (and potentially buckets) to perform distributed resource management.

All of the physical entities offer interfaces to operate with the world. The word and letter stations have controls to move letters and will block further actions until the current actions are complete. As the bucketbots have richer interactions with the environment, the bucketbot base class has more functionality. The bucketbot base interfaces include functionality to accelerate and stop at a specified point, accelerate until maximum speed is reached, turn to a specific angle and notify when the turning is complete, grab and release a bucket, and find bucketbots and buckets within a vicinity. The bucketbot base class also contains a base task system.

ALPHABET SOUP also has a waypoint implementation which may be utilized and extended to constrain bucketbot motions and bucket storage to an arbitrary graph. It is particularly useful as the number of buckets and bucketbots scale up, as it aids in managing navigation and defining coordinated paths or highways.

Research Challenges in Alphabet Soup

ALPHABET SOUP contains many challenging topics for further study. While all of the problems are interrelated, most of them can be abstracted to either architectural or resource management issues. Among the architectural issues is the dichotomy between a system with centralized or decentralized control. ALPHABET SOUP is an excellent environment in which one can study the tradeoffs between the two approaches. In this section, we highlight some of the research problems, and follow it with a discussion of how decentralized market-based solutions could be employed to address the research problems.

Among the first questions to address is how many buckets are needed and how they should be arranged on the floor. One can imagine neat, orderly rows of buckets, with pathways for the bucketbots to travel when burdened with a bucket. One can also imagine dense blocks of storage that entail a *tile problem* in order to extract the inner buckets (Gue 2006). It is easy to imagine the warehouse laid out on a grid, but because the buckets in ALPHABET SOUP are round, non-linear packing choices are also an option. Further, the layout need not be fixed; instead, it could adapt to the patterns of word creation and bucketbot motion.

The lowest level of coordination is among the bucketbots moving on the warehouse floor. Although the bucketbots are entirely predictable, coordinating their motion to prevent collisions and congestion is a challenge. Controlling the motion of the bucketbots could be done by a central planner, or it could be done through peer-to-peer communication.

As we move into higher levels of abstraction, we find several key resource allocation issues. Foremost, is the problem of task assignment. On the receiving side, when do letters need to be put into inventory, and which bucketbot(s), bucket(s), and station will be chosen to accomplish the task? Similarly, when a word needs to be built, the bucketbot(s) and bucket(s) need to be scheduled for deliveries to a station. The dynamic nature of the system leads to challenging research questions in the areas of queueing theory and scheduling, and the large number of degrees of freedom admit a wide variety of solutions.

To illustrate the complexities of these issues, consider bucketbot A, which may be close to half-empty bucket B and to station S. When letter L needs to be stored, it could be put into bucket B. A may be the closest free bucketbot, but, bucketbot D is setting down a bucket right next to B, and will be free to grab B in a moment. Which bucketbot should be assigned the task? Now consider the case where the letter to be put away is a 'u', and bucket C has a 'q'. Although C is farther away than B, it may be worth the effort to bring it to station S because of the increased likelihood that 'q' and 'u' will be pickable at the same time.

Similarly, when building words, bucket E may have two letter tiles needed, while buckets F and G may have only one, but may be much closer. Which is the better allocation? Further, when it is time to assign the word, there may be more than one station that could do the job, and the best choice of station may be dependent on the proximity of the letter tiles needed for that word. One's ability to optimize these types of decisions will depend upon how dynamic the environment is. In some real-world situations, all of the jobs are known the night before, while in companies with same-day delivery, the jobs are dropping on the warehouse in real time.

Potential Auction-Based Solutions

Because the primary problems in ALPHABET SOUP are based on resource management, it is a prime ground for testing auction-based resource allocation strategies in real-world warehouse management problems. Although the ALPHABET SOUP warehouse is a cooperative environment, there may be benefits to decentralizing aspects of the decision making, particularly if the bucketbots are relatively autonomous. A suitable "currency" would need to be created for the market economy, with either energy or time being a natural first step.

One market-based approach would be for stations to bid on jobs while subcontracting the letter tile delivery to bucketbots who contract with buckets. This approach would create interesting task dependency networks (Walsh & Wellman 1998). The ContractNet protocol (Davis & Smith 1983; Sandholm 1993) is a natural approach to attempt.

Alternatively, word stations could employ combinatorial auctions as a means of obtaining letters. The nature of the allocation problem is combinatorial because a word consists of a certain number of letter tiles, and the system prefers the cheapest solution to the entire word. A closer bucket may be passed up if the only free bucketbot in the area is needed for a different bucket.

A different approach would be to assign tasks in an arbitrary or round-robin manner and let a market *re-allocate* the assignments. Based on this initial allocation, bucketbots, stations, and buckets could auction off their tasks, and choose to perform a task when it is most profitable. Bucketbots, buckets, and stations could gain compensation for both the completion of tasks and from selling tasks, evaluating the utility of having each task based on how much utility it would gain versus expend from completing the task.

Determining when to hold task assignment auctions and which entities to include is also an important issue. With hundreds of open tasks, hundreds of buckets and bucketbots to perform those tasks, and allocation efficiency being dependent on combinatorial effects, the bidding space is too large to be tractable. To solve this problem, some heuristics are needed to limit participation in auctions. Using physical locality for gathering participation for an auction and propagating task information might offer some usefulness. However, it will not help cases when two buckets are far apart but one could accomplish the other's task more efficiently. Rather, adding some other metric of similarity would be more useful, such as using cosine similarity on bucket contents to group buckets for auctions based on their ability to accomplish similar tasks.

An interesting research direction is evaluating how the choice of bidders and resources affects throughput. Given the numerous ways of applying auctions to ALPHABET SOUP, which bidder and resource choices most improve throughput, and are any seemingly different auction resource management implementations functionally equivalent?

Other Potential Solutions

While centralized planning can make optimized solutions more straightforward to obtain, many of ALPHABET SOUP's central optimization problems are NP-hard. This level of computational complexity does not scale well with purely centralized or exhaustive solutions with near-realtime demands. Myopic best-first techniques, as well as traditional planning techniques, may prove useful either in terms of task assignment or in bucketbot motion planning.

Related Work

Large scale, multi-robot systems have been used to solve problems such as search and surveillance (Konolige *et al.* 2004) and assembly (Simmons *et al.* 2002). To the best of our knowledge, ALPHABET SOUP is the first testbed for multi-robot warehouse and physical distribution/routing problems.

ALPHABET SOUP has a higher-level focus than most other robot simulators, as its goal is to provide a framework for studying resource allocation in physical routing. Frameworks such as Player/Stage (Collett, MacDonald, & Gerkey

2005) and CARMEN (Montemerlo, Roy, & Thrun 2003) focus on robot sensing capabilities, localization, and environment discovery, whereas ALPHABET SOUP resides in a highly controlled environment which fosters ease of position determination and communication. Other simulators do not easily support a dual-layered environment where robots can pick up buckets and freely drive above or beneath them, without adding computationally costly 3D environments.

Market-based and auction control techniques are an effective resource allocation method in multi-agent systems (Wellman & Wurman 1998), and have been implemented in many different capacities and environments (Gerkey & Matarić 2002; Dias *et al.* 2004; Simmons *et al.* 2002). As it contains resource allocation problems, ALPHABET SOUP is a particularly good candidate for auction-based approaches.

ALPHABET SOUP is also a useful model of a practical problem for validating robot motion planning techniques, such as those devised by Clark (2005). Likewise, ALPHABET SOUP is valuable for studying more general distributed coordination techniques including those surveyed by Jennings (1996) and that implemented by Parker (1998).

As testbeds for multi-robot control make it easier to explore high level algorithms, they have been built for many other problems as well. One of the authors implemented a software testbed for "Capture the Flag" style coordination robot games (D'Andrea & Babish 2003). Hardware testbeds are useful for investigating real-world complications that are not always obvious in simulations, such as the Caltech multi-vehicle wireless testbed (Cremean *et al.* 2002).

Conclusions and Future Work

We present ALPHABET SOUP as a model of emerging robot-assisted warehouses. The model captures many of the key coordination and allocation challenges faced in real systems, but does so at a level of abstraction that facilitates study. The ALPHABET SOUP platform includes a detailed model of bucketbot behavior and realistic work profiles. It is also highly configurable, which allows researchers to direct their studies at particular aspects of warehouse management.

We hope the platform will be of use to researchers studying multi-agent systems, resource allocation, vehicle coordination in MMVS, and operations research.

Acknowledgments

The Kiva System is the fruit of the labor of many people. The authors are indebted to them for creating the opportunity to work on such an interesting project.

References

Clark, C. 2005. Probabilistic road map sampling strategies for multi-robot motion planning. *Journal of Robotics and Autonomous Systems* 53(3-4):244–264.

Collett, T. H.; MacDonald, B. A.; and Gerkey, B. P. 2005. Player 2.0: Toward a practical robot programming framework. In *Proceedings of the Australasian Conference on Robotics and Automation (ACRA 2005)*.

Cremean, L.; Dunbar, W. B.; van Gogh, D.; Hickey, J.; Klavins, E.; Meltzer, J.; and Murray, R. M. 2002. The caltech multi-vehicle wireless testbed. In *Proceedings of the 41st Conference on Decision and Control*.

D'Andrea, R., and Babish, M. 2003. The RoboFlag testbed. In *American Control Conference*, 656 – 660.

Davis, R., and Smith, R. G. 1983. Negotiation as a metaphor for distributed problem solving. *Artificial Intelligence* 20:63–109.

Dias, M. B.; Zinck, M.; Zlot, R.; and Stentz, A. T. 2004. Robust multirobot coordination in dynamic environments. In *IEEE International Conference On Robotics And Automation*, volume 4, 3435–3442.

Gerkey, B. P., and Matarić, M. J. 2002. Sold!: Auction methods for multirobot coordination. *IEEE Transactions On Robotics And Automation* 18:758–768.

Gue, K. R. 2006. Very high density storage systems. *IIE Transactions* 38(1):79–90.

Jennings, N. R., and Bussmann, S. 2003. Agent-based control systems: Why are they suited to engineering complex systems? *IEEE Control Systems Magazine* 61–73.

Jennings, N. R. 1996. Coordination techniques for distributed artificial intelligence. In O'Hare, G. M. P., and Jennings, N. R., eds., *Foundations of Distributed Artificial Intelligence*. Wiley. 187–210.

Konolige, K.; Fox, D.; Ortiz, C.; Agno, A.; Eriksen, M.; Limketkai, B.; Ko, J.; Morisset, B.; Schulz, D.; Stewart, B.; and Vincent, R. 2004. Centibots: Very large scale distributed robotic teams. In *Proceedings of the International Symposium on Experimental Robotics*.

Montemerlo, M.; Roy, N.; and Thrun, S. 2003. Perspectives on standardization in mobile robot programming: The Carnegie Mellon Navigation (CARMEN) toolkit. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003)*, volume 3, 2436–2441.

Parker, L. E. 1998. Alliance: An architecture for fault tolerant multirobot cooperation. *IEEE Transactions On Robotics And Automation* 14(2):220–240.

Sandholm, T. 1993. An implementation of the contract net protocol based on marginal-cost calculations. In *Proceedings of 11th National Conference on Artificial Intelligence (AAAI-93)*, 256–262.

Simmons, R.; Smith, T.; Dias, M. B.; Goldberg, D.; Hershberger, D.; Stentz, A.; and Zlot, R. 2002. A layered architecture for coordination of mobile robots. In Schultz, A., and Parker, L., eds., *Multi-Robot Systems: From Swarms to Intelligent Automata*. Kluwer.

Walsh, W. E., and Wellman, M. P. 1998. A market protocol for decentralized task allocation. In *Third International Conference on Multi-Agent Systems*, 325–332. citeseer.csail.mit.edu/walsh98market.html.

Wellman, M. P., and Wurman, P. R. 1998. Market-aware agents for a multiagent world. *Robotics and Autonomous Systems* 24:115–25.