

High- and Low-Level Planning for Mobile Agents

Kamilah Walker, Hayward Little, Albert Esterline

Computer Science Department/NASA ACE

North Carolina A & T State University

Greensboro, NC 27411

Email: {kkwalker, hayward, esterlin}@ncat.edu

Abstract

We handle motion planning for a multi-agent system at both a low level, where metric details of the paths are worked out, and a high level. At a high level, agents are associated into groups that more or less travel together and a hierarchy of levels allows motion planning to be modularized and to be simplified over large areas. A mobile agent is an autonomous entity that travels from a starting point to a destination point without human intervention or collision. We address both static environments, where all obstacles are stationary, and dynamic environments. In a dynamic environment, a path must also be specified as a function of time. Here “motion” denotes a path with timing information. We show how to represent multi-agent plans at a high level using the statechart formalism. We emphasize an explicit approach for representing aggregate plans. The assignment of agents to roles in a plan and agents as position holders are also discussed. Finally, our work is related to philosophical societal theories.

Keywords: motion planning, mobile agents, visibility graph, multi-agent system, accessibility graph, statecharts, societal theories, joint intention, mutual belief, agreement

INTRODUCTION

Incorporated in this paper are both the high- and low-level aspects of multi-agent path planning. The high-level component includes using statecharts to model a multi-agent system along with an agent’s dependencies, actions, and roles in a system. The simpler phases of path planning are found in the low-level features, such as how to represent agents in a static or dynamic environment.

This paper relates to research conducted for the NASA-supported project “Motion Planning in a Society of Intelligent Agents.” Agents in this context can be considered as robots. Mobile agents are capable of travelling collision-free from a start point to a destination point without human intervention. We represent multi-agent plans at a high level with statecharts, which represent both hierarchies of states (hence abstraction hierarchies) and orthogonal (concurrent) components. Synchronization among components could be achieved implicitly, by modeling the aggregate through a web of references. [1] Instead, we use an explicit approach, which makes plans more reusable and extendible. The second section introduces the statechart notation; the third section shows how statecharts are used to

represent multi-agent plans. The next section describes the explicit approach to synchronizing aggregate plan. Subplans are not the agents but rather the role positions to which agents are assigned. The following section describes in detail the place of agents as position holders. The last section which notes the high-level aspects of path planning sketches how our approach relates to central notions in philosophical societal theory, such as joint intention, agreement, and mutual belief.

Low-Level Aspects of Path Planning

We cover the low-level aspects of path planning in two contexts, static environments and dynamic environments. A static environment consists of stationary obstacles. Here determining a shortest, collision-free path from a start point to a destination point requires knowledge of all pairs of points visible from each other (without an intervening obstacle), so a visibility graph [2] is created. Dynamic environments, however, pose a more difficult challenge. Since the obstacles are in motion, the initial path may no longer be collision-free after a certain period of time. This problem is addressed by using an accessibility graph [2] to determine whether the current path is still valid or must be changed. Also, within this dynamic environment, it is assumed that the robot can move faster than any obstacle. (When all objects have zero velocity, the accessibility graph reduces to the visibility graph.)

High-Level Aspects of Path Planning

Multi-agent systems can be viewed as collections or societies of intelligent agents. Societies of agents may collaborate to achieve some common goal. The central assumption is that more can be achieved by working together than by working alone. This assumption suggests the use of philosophical theories of societal interaction in modeling the operation of a mobile agent society. The agents use motion plans to accomplish their societal mission [1].

The key to understanding group attitudes and actions is the notion of norms. Deontic logic, as a logic of norms, has been applied to motion planning to deal with agent violation behavior and to restore a system to an ideal situation [3] However, given a society of agents, there must exist interaction, coordination and the threat of collision among the agents. To handle these situations, the concepts of agreement, joint intention, joint action, and mutual belief from societal theories [4] are applied. Furthermore, common knowledge is a prerequisite of agreement and coordination; therefore, applying common knowledge to the systems seems necessary.

Since multi-agent motion planning is within the realm of reactive systems, statecharts are used to represent the structure of multi-agent activity because of their ability to express hierarchy, concurrency, and communication in the behavior of complex reactive systems. As mentioned before, however, to represent the structure of systems with interaction, coordination, and collision, statecharts must be enhanced with deontic logic [3], epistemic logic, societal theories, and some additional structure changes.

STATECHARTS

Statecharts [5] are a visual formalism to specify the behavior of complex reactive systems. They extend conventional state-transition diagrams to include the notions of hierarchy, concurrency (orthogonality), and communication. Statecharts also may be viewed at varied levels of detail through abstraction and refinement. Thus,

Statecharts = state-diagrams + depth + orthogonality + broadcast-communication

Hierarchy in statecharts is achieved by grouping states into superstates. This is a *XOR* (exclusive-or) decomposition, as shown in Figure 1. This statechart includes two states, A and B. A has two substates, C and D. If the system is in A, then it must be either in substate C or D, but not both (exclusive-or relation). The arrow indicates the transition direction; the label over it indicates the transition event. Suppose that the system is in state B. When event *e* occurs, it goes directly to substate C. When event *h* occurs, it goes to A as a whole (the arrow labeled “*h*” stops at the boundary of A). The unlabeled dotted arrow to substate D of A indicates that D is the default state of A. So, the system goes to D on event *h*. When event *f* occurs, whatever substate of A the system is in, it goes to B (the arrow labeled “*f*” starts at the boundary of A). Finally, if the system is in substate C, when event *g* occurs, it goes to D if and only if condition *c* holds (see the “*c*” in square bracket after the label “*g*”)

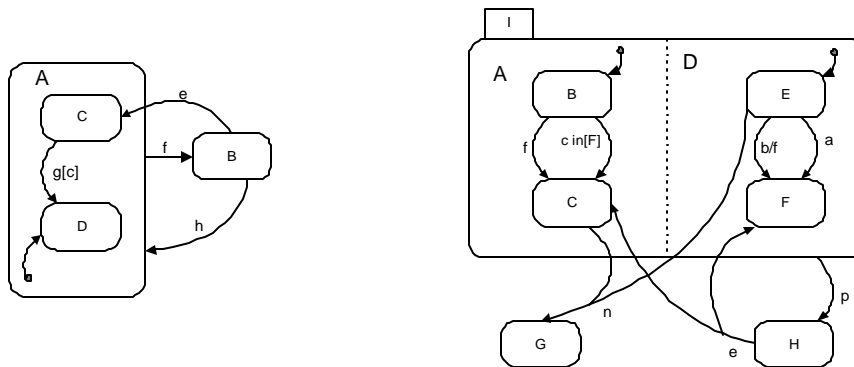


Figure 1 Statechart with *XOR*-state

Figure 2 Statechart with *AND*-state

Concurrency in statecharts is shown through orthogonality, which is an *AND* decomposition captured by the partition of the statechart as in Figure 2. There are three states, G, H and I. I has two orthogonal components, A and D (see the dashed boundary). A and D also have *XOR* decompositions so that, if the system is in I, it must be in one of four possible pairs of substates $\langle B, E \rangle$, $\langle B, F \rangle$, $\langle C, E \rangle$ or $\langle C, F \rangle$. If the system is in the state H, when event *e* occurs, it enters I and ends up at substates C of A and F of D. If the system is in substate E of D and either B or C of A, then, when event *a* occurs, it goes to F and stays in whatever substate of A it was in. However, when event *b* occurs, a transition labeled *b/f* is taken, the action *f* is immediately activated and is regarded as an (internal) event. Now, if the system is also in the substate B of A, then it goes to C; thus, the system goes to F and C when event *b* occurs. If the system is in substates C and E in I, when event *n* occurs, it leaves the state I and enters state G. The

arrow labeled “*p*” indicates that, if the system is in any one of four pairs of substates of I aforementioned, when *p* occurs, it goes to state H.

The communication mechanism in statecharts is based on broadcast, whereby the sender may proceed even if nobody is listening. It also forces all enabled listeners to accept and respond to the message simultaneously.

PLANS REPRESENTED BY ENHANCED STATECHARTS

We give an example that represents with a statechart a plan for planting a plant. Figure 3 illustrates the dependency relations among the following steps: **1.** Dig hole. **2.** Put some soil back into the hole. **3.** Mix soil conditioner with the soil in the hole. **4.** Mix soil conditioner with the remaining soil out of the hole. **5.** Check planting depth; fill to planting depth. **6.** Pour in water with dissolved fertilizer. **7.** Take the plant out of the pot. **8.** Put the plant into the hole. **9.** Put the rest of the soil in the hole around the plant. **10.** Pour the rest of the water/fertilizer around the plant. **11.** Put mulch around the plant.

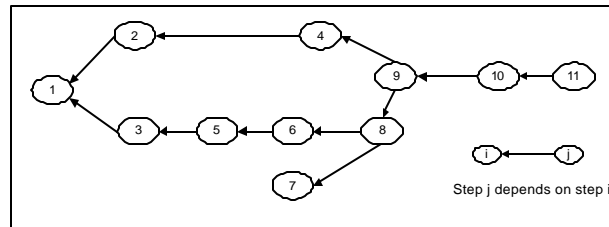


Figure 3 Dependencies Involved in Planting a Plant

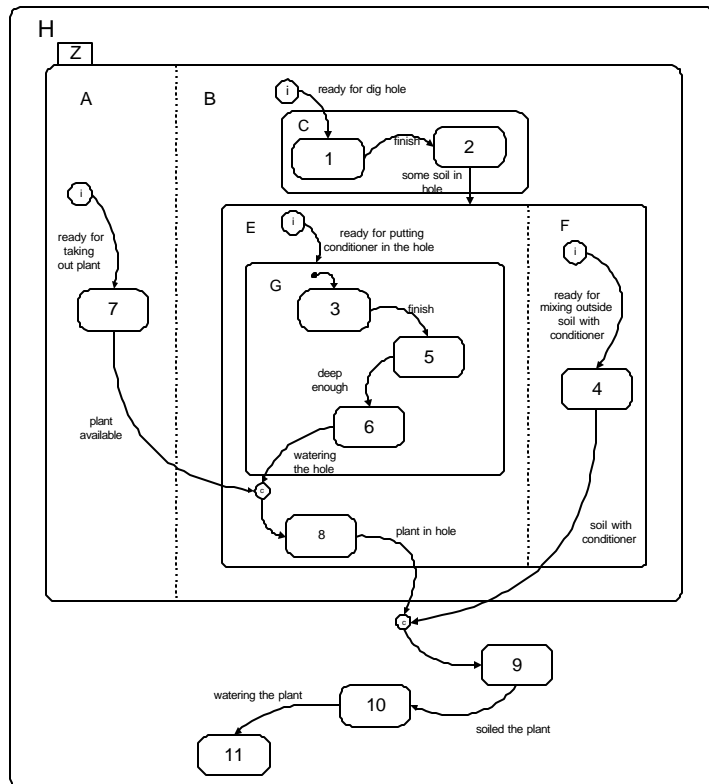


Figure 4 Plan-Statechart for Planting A Plant example,

From this we derive a plan represented by the statechart shown in Figure 4. This statechart is a *plan-statechart*. We introduce an idle state in our plan-statecharts, a circle with an *i* inside. Whenever a state with an idle substate is entered, the idle state is a default state. We can decompose a plan into several smaller subplans carried out sequentially (*XOR*) or concurrently (*AND*). We also can group plans into a superplan hiding details. A circle with a *c* inside indicates joint conditions for a transition. In Figure 4, we omit the action part in the label of a transition, since it is simply the inception of the ensuing activity.

Concurrency among the plans in a plan-statechart can be shown clearly and flexibly. For

plan *Z* in Figure 4 has two orthogonal subplans, *A* and *B*, which can proceed in parallel except that subplan 7 of *A* must be carried out before the subplan 8 of *B*. Because we introduce the idle state, when the subplan 7 of *A* occurs is not constrained related to *C*, *G*, and *F* – so maximum concurrency is portrayed.

WHOLE-PARTS CHARTS FOR CONCURRENCY AND SYNCHRONIZATION

Traditionally, interactions among orthogonal components are synchronized implicitly, that is, different components of an *AND*-state synchronize with each other by the combined effects of the broadcast and triggering mechanisms [6]. These implicit methods lack reusability, understandability and extendibility because implicit synchronization does not encapsulate the details of components of an *AND*-state. [6] proposed an explicit method that extracts synchronization aspects from the statechart and encapsulates them into an additional statechart, representing the whole. The *whole* section and *parts* sections evolve concurrently as in an *AND*-state. The following communication constraints enforce self-containment and encapsulation: (1) Events cannot trespass the boundary among the different *parts* sections. (2) Events can be broadcast both from the *whole* section to the *parts* sections and vice versa. So the *whole* knows its *parts* but the *parts* ignore each other. Synchronization aspects can be expressed entirely in the *whole* section.

In plan-statecharts, a *whole-parts* chart is associated with any *AND*-plan involving interactions among its orthogonal components. A whole-parts chart is a rectangle with two sections, a parts section and a whole section, as illustrated in Figure 5. The interacting components are explicitly expressed as parts as rectangles separated by solid lines to emphasize rule (1). The events which control the synchronization among the parts are listed in the parts. The whole component is an additional entity to show explicitly how the parts are synchronized. A dash line separates the whole component from the parts to emphasize rule (2). A solid circle in the whole component indicates the start of the whole, and a solid circle within a circle indicates the termination of the whole.

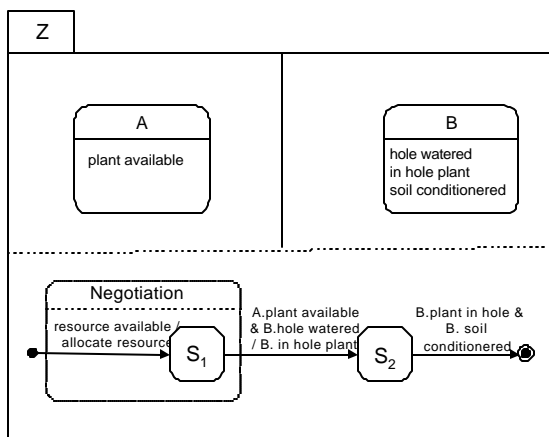


Figure 5 The Whole-Parts Chart of An *AND*-Plan *Z*

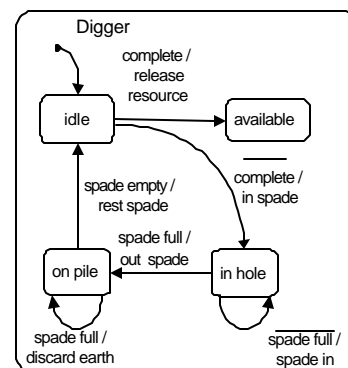


Figure 8 The Position-related Plan-Statechart for Digger

Figure 5 is the whole-parts chart for *AND*-plan *Z* in the figure 4. Two orthogonal components, *A* and *B*, of *Z* are two parts in this chart. The synchronizing events are listed in the parts. A prefix notation indicates the source of events, so that, for example, *A.plant available* means the *plant available* event comes from plan *A*. The *Negotiation* part in the

whole component shows how to allocate the resources. The Z plan as a whole will advance from state S_1 to S_2 if $A.plant\ available$ and $B.hole\ watered$ are true. This transition also initializes of S_2 's activity $B.in\ hole\ plant$. The occurrence of events $B.plant\ in\ hole$ and $B.soil\ conditioned$ leads to the exit from Z.

AGENTS AS POSITION HOLDERS IN PLAN-STATECHARTS

When a plan is decomposed at a deep level, the subplans show plans for role positions, not agents. This allows flexibility in the agent assignment process. For example, the *dig-hole* plan in Figure 4 (subplan1) is an AND-plan with two parts, diggers and a hole. At this level, how the actions are to be performed is shown. Therefore, we use the position-related plan-statecharts to show the position holder's plan. Figures 6 and 7 are the plan-statecharts for the *digger1* and *hole* position holders.

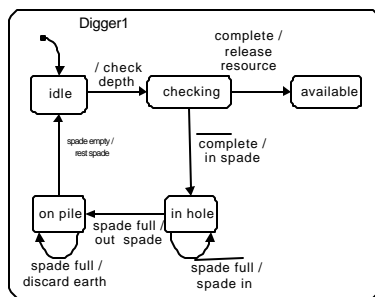


Figure 6 The Position-Related Plan-Statecharts for Digger1's Plan

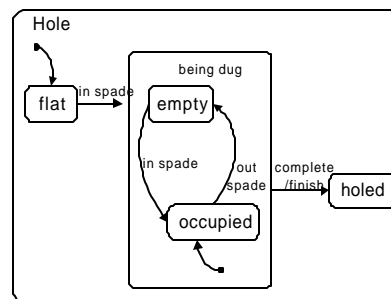


Figure 7 The Position-Related Plan-Statechart fro Hole's Plan

The *dig-hole* plan may involve more than one digger. The other diggers' positions (called *digger*) differ from the position *Digger1* by lacking a checking step. Figure 8 shows the general digger's position plan. When several diggers are assigned to the *dig-hole* plan, the actions among the diggers must be synchronized to avoid their spades colliding. Figure 9 shows two diggers' whole-parts chart for *dig-hole*. The negotiation part in section *whole_dig-hole* assigns two agents to the positions *digger1* and *digger* and initiates the plan.

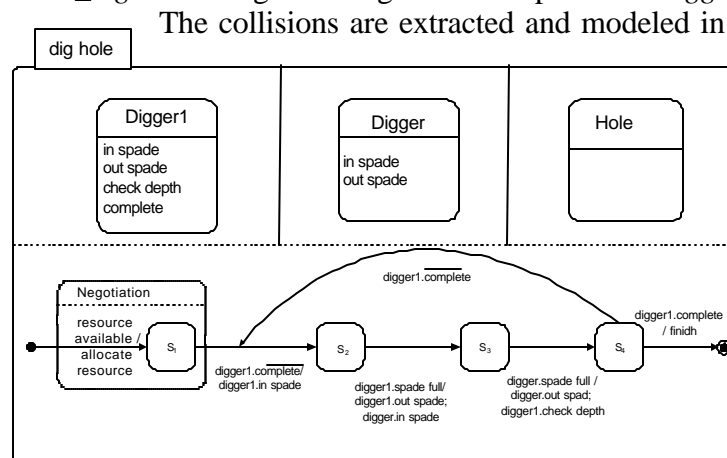


Figure 9 The Whole-Parts Chart for Dig-Hole Plan

The collisions are extracted and modeled in the *whole* section, which explicitly expresses the synchronization among the components.

Explicitly expressing the synchronization aspects entirely in a *whole* section provides a natural context for extensibility. We can add one more digger in the *dig-hole* plan without any change to the parts already present and by including just one additional state in the whole.

SOCIETAL THEORY FOR MULTI-AGENT SYSTEMS

The bridge between a plan-statechart and a multi-agent system is the negotiation process in the whole-parts chart associated with every *AND*-plan. Agents in a multi-agent system use negotiation for conflict resolution and coordination. [7] Based on societal theory [4], agents involved in a plan can be viewed as a social group with task and role structure, and this structure is characterized by social norms. A plan is formed based on agreements within an authority system. The plan consists of the positions, which need to be filled by the executive agents, the subplans or tasks, which the position holders should carry out, and the rules, which govern tasks and duties that agents have in their positions. When the agents in a social group fill the positions in a plan, they not only obtain knowledge about their own subplans and the rules, but also know that other agents know their subplans and rules. When an agent agrees to be a position holder, he also agrees to follow his subplan and the rules. Therefore, the agents, as position holders of a plan, not only form a joint intention but also an agreement to jointly perform the plan. Furthermore, the agents must mutually believe that they made such an agreement, and the agents must commit to doing their parts of the job. Mutual belief is like common knowledge [8] in a group except that it is generally future directed.

After the negotiation process in the *whole* section of a whole-parts chart, the bridge between a multi-agent plan and a multi-agent system is established. The joint intention formed in the negotiation process can serve to initiate joint performance for a plan, and the agreement will guide the agents' behavior and effect coordination.

LOW-LEVEL ASPECTS

Before we can begin modeling multi-agent systems, we must understand the simpler concepts of path planning, specifically, the low-level aspects.

Abstractions

We begin by listing the class abstractions involved in implementing the behavior of both the static and dynamic environments. Our abstractions (classes) include the following: *Point*, *Line*, *Polygon*, (graph) *Node*, *Binary Search Tree*, *Priority Queue*, and *Undirected Graph*. Both *Binary Search Tree* and *Priority Queue* are used for searching in the line intersection function. For the shortest path, an adjacency structure is created to implement the *Undirected Graph*. A *Point* holds x and y coordinates, along with a name field. A *Line* holds the index values of two *Points* and a name field. A *Polygon* contains the number of edges, a linked-list of the *Points* in counterclockwise order, and a name field. Our *Binary Search Tree* refers to the index of a *Point* and its x coordinate. The *Priority Queue* contains elements, each with an index field and y coordinate field. And finally, our *Undirected Graph* stores a set of *Nodes*; its edges connect pairs of *Nodes*.

Obstacles

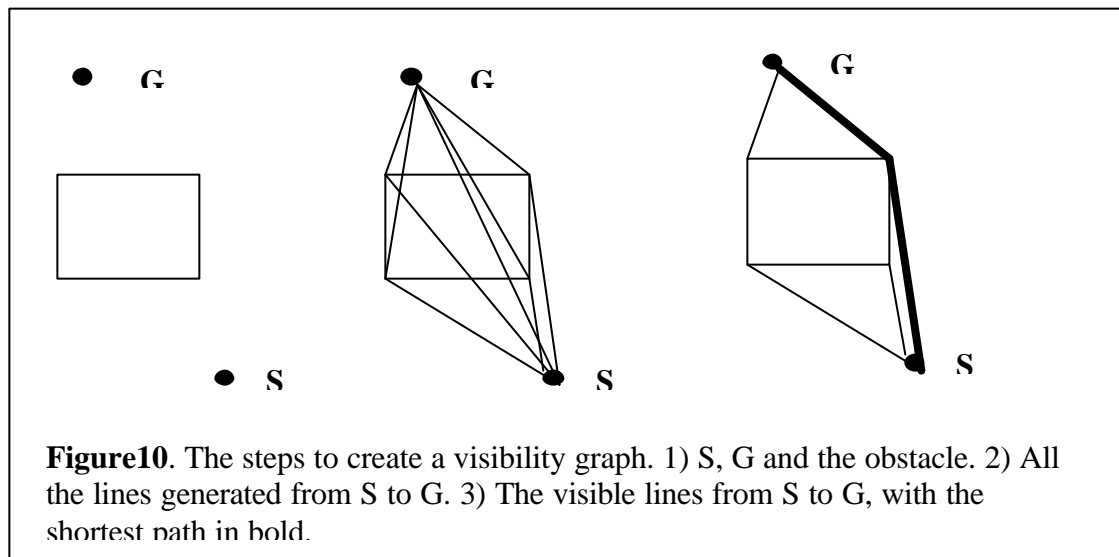
Obstacles are assumed to be convex polygons [9] Convexity is checked using the convex hull algorithm implemented is Graham's scan. [10] If all the points of the polygon are on the convex hull, then the test is successful. To ensure that no polygons overlap, a search is done to ensure that the endpoints of polygons do not fall in other polygons.

Visibility Graph

The visibility graph (see Figure 10) is created by:

1. Read in start, destination, and the obstacles.
2. Generate all paths possible from the start to the destination.
3. Remove those paths obstructed by obstacles.
4. Determine the shortest path.

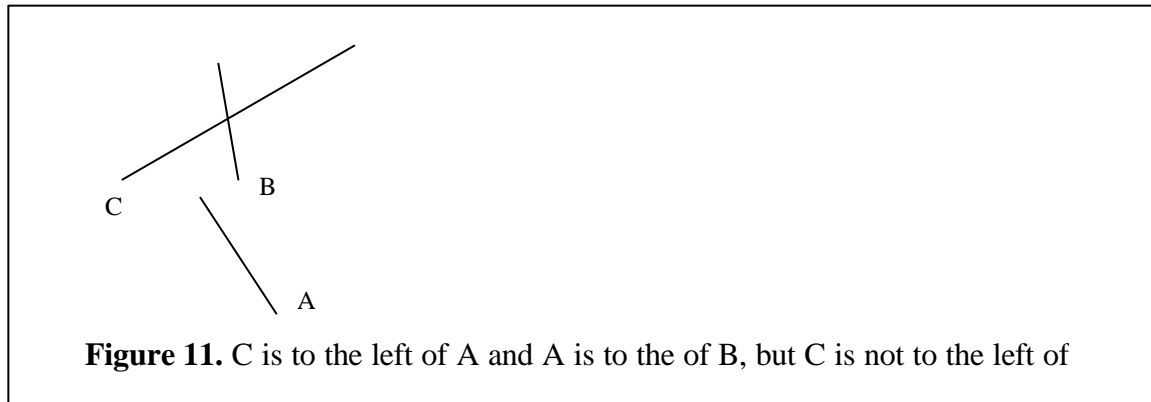
An adjacency structure contains points that are used to represent the visibility graph. We implemented Dijkstra's algorithm to find the shortest path.



Line Intersection

To create the visibility graph, all of the points need to be connected together. This is done with a brute force method done in $O(N^2)$ time, where N is the number of points. Once we have an initial graph, we must remove any lines that intersect with a polygon. We accomplish this by scanning through the graph in a bottom-to-top method. In order to simulate scanning in lines, we use a priority queue, binary search tree and linked-list. The binary search tree is used to insert/delete the points of the lines in a bottom-to-top fashion, while the linked-list stores intersections found within the tree. By using a binary search tree, as opposed to nested for loops, our search algorithm runs in $O(M \log_2 M)$ versus $O(M^2)$ time, where M is the number of line segments (normally $O(N^2)$). To ensure that the lowest remaining point is used for insertion/deletion, a priority queue is used. Inserting all the ordinates of the lines generates the priority queue (which is ordered in non-decreasing order of the ordinates). Insertion of a line into the binary search tree is done when the lowest point of a line is inserted. Once the line's other endpoint is found, it is deleted from the tree.

Since our binary search tree is based on line orientation, we use a counterclockwise function to determine if a line should be in the left subtree or right subtree of another line in the tree. This function also detects whether the two lines intersect. Necessary conditions must be met to verify the tree is properly shaped after a line is removed. This is because, once a line is removed, it is not guaranteed that its children can be linked to the parent node of the line deleted, since the generalized comparison operation is not transitive (see Figure 11)¹⁰ For these cases, the children are reinserted into the subtree.



Dynamic Environment

An accessibility graph is a generalized visibility graph. If all the obstacles in a dynamic environment had a zero velocity, then the accessibility graph would reduce to a visibility graph. Basic assumptions are that our agent is a point and moves faster than all the obstacles. These assumptions are necessary to create time-minimal motion, which can be represented as a sequence of edges in the accessibility graph [2]. A path in a dynamic environment is known as a *trajectory*, which refers to velocity and direction along a path. It is essential that our agent is capable of developing *destination points* within a specific time frame. The period of time during which our agent is able to move to the destination points without collision is the *accessibility time*. These destination points become various steps necessary to reach our final goal.

The following steps apply to constructing accessibility graphs for dynamic environments. (1) Insert the starting vertex into a priority queue. (2) The vertex V with the youngest associated accessibility time is removed from the queue. Each vertex, at the time and place it is reached, is added to the accessibility graph, and it, in turn, is used as the source for a set of linear motions, which lead to a set of new vertices for the graph. All obstacles are now considered to be where they would be when the robot reaches V . The accessible vertex set with respect to V is the set of all (obstacle or destination) points accessible from V , that is, which the robot can meet after leaving V , without prior interception by any (moving) obstacle. (3) If the vertex V dequeued in step (2) is the destination point, then the motion is recorded; otherwise, all other vertices in the accessible vertex set for V are inserted and steps 2 and 3 are repeated for the newly inserted vertices. When the destination is reached, we are guaranteed to have a minimum-time path from the start point.

INTEGRATING HIGH- AND LOW-LEVEL ASPECTS OF PATH PLANNING

In modeling a multi-agent system, the high-level aspects of path planning are a more abstract representation of the low-level aspects. When utilizing statecharts to model dynamic and static environments, the metric details are ignored, and, instead, the focus turns more to the synchronization of the objects. Most of the high-level features of path planning involve the synchronization of multiple agents with multiple goals.

Presence at each start and destination point for an agent can represent a state. So, when there are several successive goals, a statechart can best model successive, state-to-state transitions. The metric properties of the motions of objects become inconsequential, and their interactions with each other become more important. In a hierarchical planning strategy, the dependencies among successive goal states are captured by a statechart.

Planning could be handled by a causal-link partial-order planner that searches through the plan space. The planner would begin with an initial plan representing just the beginning and end steps, and on each iteration the planner would add one more step. If this leads to a bad plan, it would backtrack and try another path in the search space. The planner would focus its search by only adding steps that would serve to achieve a precondition that has not yet been achieved. For example, in planting a plant the dependencies for other steps serve as their preconditions. When the precondition is satisfied, then the plan is able to move closer to completion.

An advantage of using high-level features to model multi-agent systems versus modeling single agents using low-level features is that plans can be reused. Also, among the advantages of using high-level features to model multi-agent systems is extensibility. This allows the system to accommodate more complex elements in the system.

CONCLUSION

We have shown how statecharts are used to represent multi-agent plans and have emphasized an explicit approach to modeling aggregate plans, explicitly representing the whole plan. This approach makes plans more reusable and extensible. It is important that agents are not represented by parts of statecharts but rather are assigned such parts. For one thing, this allows a useful notion of the state of an agent, namely, a pair consisting of the history of the agent and a set of pointers to the parts of his plan currently being executed. The hierarchical nature of statecharts allows an agent's history to be summarized by remembering only higher-level states when details are not necessary. Given this notion of a state, a modal analysis of multi-agent systems [8] is possible. In particular, the key notion of common knowledge is available, and, indeed, our analysis supplies structuring for groups and histories that enhances the application of this notion. Our statechart-based approach also supports a deontic analysis of agent actions in that transitions can be seen as obligated actions. A plan-statechart could be extended with violation transitions and states, which are thus forbidden but may actually occur – allowing for fault analysis. [11] Finally, these modal notions lead into the larger topic of societal theory, only touched on in this paper.

By developing the necessary classes to generate visibility and accessibility graphs, it is possible to form plans for agents to travel from a start point to a destination point in static and dynamic environments. Our software was implemented in Java. By taking an object-oriented approach, we have code that can be efficiently reused and utilized in future projects.

In modeling multi-agent systems, we emphasize the abstraction of the low-level aspects of path planning into discrete states in a statechart. In addition, integrating the high- and low-level aspects of motion planning allows us more flexibility in modeling real world situations where change is constant.

REFERENCES

1. Campbell, James. "Modeling a Society of Agents Using Societal Theories," Masters Thesis, Dept. of C.S., North Carolina A & T State University, Greensboro, NC, (1998).
2. Fujimura, K., *Motion Planning in Dynamic Environments*. Tokyo: Springer-Verlag, (1991).
3. Campbell, J. and Esterline A. "Motion Planning in a Society of Mobile Agents," Proc. NASA-URC '98 Conference, Huntsville, AL (1998).
4. Tuomela, Raimo., *The Importance of Us: A Philosophical Study of Basic Social Notions*, Stanford, CA: Stanford University Press, (1995).
5. Harel, David., "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming* 8, 231-274, (1987).
6. Pazzi, L., "Extending statecharts for representing parts and wholes." *Proceedings of the 23rd EUROMICRO Conference*, (1997)
7. Müller, J. H. "Negotiation Principles." in G.M.P. O'Hare and N.R. Jennings (eds), *Foundations of Distributed Artificial Intelligence*, New York: Wiley, 211-229, (1996).
8. Fagin, R., Halpun, J. Y., Moses, Y., Vardi, M. Y., *Reasoning about Knowledge*, Cambridge, MA: The MIT Press, (1995).
9. Preparata, F. and Shamos, M., *Computational Geometry: An Introduction*. New York: Springer-Verlag, (1985).
10. Sedgewick, Robert, *Algorithms in C*. Reading, MA: Addison-Wesley Publishing Company, (1992).
11. Wu, Xiaohong. "The Formal Representation of Multi-Agent Plans," Masters Thesis, Dept. of C.S., North Carolina A & T State University, Greensboro, NC, (1999).