Conquering Large Zones by Exploiting Task Allocation and Graph-Theoretical Algorithms

Chengqian Li

Dept. of Computer Science,
Sun Yat-sen University
Guangzhou 510006, China
lichengq@mail2.sysu.edu.cn

Abstract. The Multi-Agent Programming Contest is to stimulate research in the area of multi-agent systems. In 2012, for the first time, a team from Sun Yat-sen University, Guangzhou, China, participated in the contest. The team is called AiWYX, and consists of a single member, who has just finished his undergraduate study. The system mainly exploits three strategies: strengthening action preconditions, task allocation optimization, and surrounding larger zones with shorter boundaries. With these strategies, our team is able to conquer large zones as early as possible, optimize collaboration, and ensure efficiency. The system was implemented in C++, and in this paper, we will introduce the design and architecture of AiWYX, and discuss the algorithms and implementations for these strategies.

Keywords: multi-agent system, distributing algorithm, task allocation optimization.

1 Introduction

The Multi-Agent Programming Contest (MAPC) [1,2] is held annually, in order for researchers to deepen the understanding about the cooperations and competitions among rational agents and also develop some powerful strategies in such environments. This year, for the first time, a team from Sun Yat-sen University, Guangzhou, China, participated in the contest. The team, called AiWYX, reached the fifth place in the contest. It consists of only one member: the author of this paper. I have just obtained my Bachelor degree and am now a PhD candidate. I am a member of the knowledge representation and reasoning group led by Professor Yongmei Liu. My motivation in participating in this contest was to gain experiences in designing multi-agent systems in order to facilitate my future research in this area. These years I am actively involved in the ACM International Collegiate Programming Contest (ICPC, see `http://icpc.baylor.edu`). Before this competition I had completed an undergraduate honors thesis on Squirrel World, which was proposed by Hector Levesque as an adaptation of the Monty Karel robot world written by Joseph Bergin and colleagues in Python (see `http://csis.pace.edu/~bergin/MontyKarel`). In Squirrel World, squirrels need to move around on a two-dimensional grid and gather acorns. Squirrels

M. Dastani, J.F. Hübner, and B. Logan (Eds.): ProMAS 2012, LNAI 7837, pp. 234–244, 2013.

have both effectors (to do things in the world) and sensors (to gather information). Everything is known to the squirrels at the outset except for the locations of the acorns and some wall obstacles. The first squirrel or the first team of squirrels who gathers a certain number of acorns wins the game. I have adopted some of the strategies I developed for Squirrel World in the MAPC competition.

2 System Analysis and Design

I took part in the contest using the language C++, without using any multi-agent programming languages. There are two reasons for this. Firstly, my background is ACM/ICPC, so I am proficient in this language which is well-known for its efficiency and I did not program in Java which is not so efficient. Secondly, I did not have enough time to adapt myself to multi-agent programming languages.

We have exploited decentralization in implementing various strategies, however, the current implementation is restricted because we only deal with common knowledge [6]. When any agent's knowledge state is updated, other agents' knowledge state will be updated in precisely the same way, because of the assumption of common knowledge. Furthermore, we assume that communications between agents are perfect in this implementation. As to how to implement such strategies on a computer, we apply for a piece of main memory from the operating system, which stores the common knowledge. Hence, each agent has the same authority to access this memory space in order to communicate with other agents.

While such a team of agents is running in the competition, all agents have the goal that their team should reach a score higher than that of their rival. In any state of the world, any agent knows exactly what she should do next to achieve this goal and will start a new task immediately after completing one. In fact an agent can attain her goal by herself or through collaboration with others. Given a task, when there is only one agent intending to accomplish it, she will act by herself. However, if there are more, all such agents will collaborate to accomplish their task, that is, the task will be allocated to the agents in an optimal way. Moreover, the agents here are aggressive, that is, they keep exploring new areas of the world, never passively waiting for changes of the environment. Finally, in any state of the world, any agent is able to perform some action to achieve the goal, never lost in a dead-end.

To design and implement my system, I had spent about 250 hours. During this period, I did not discuss the design and strategies of my agent team with others, and I did not test my agents playing with other teams. I once tested my program by myself on a single computer, that is, I started a competition between two multi-agent teams, both of which were equipped with my own program. Both of them randomly selected a strategy at the beginning, thus they usually exploit different strategies, which helps me evaluate my team.

3 Software Architecture

I used C++ as the programming language, because it is so efficient and various mature data structures and algorithms are easy to code in C++. Each of the agents runs a separate program which is designed at four different levels, from the decision level to the physical level, as is described in Fig.1. Level 1 is the *decision level*, which generates an action, or applies for joining a group, according to the current state. Such a group are to accomplish a task which cannot be handled by a single agent. For example, conquering a zone is a task, which cannot be accomplished by a single agent and need a group of them. If an action is generated, the agent will herself perform it, otherwise she will join a group for coordinating the task. If more than one agent applies for the same task, the first who applies will become a manager responsible for coordinating this group of agents in an optimal way. This manager agent produces the coordination in its program architecture at Level 2 (*scheduling level*), so Level 2 is responsible for scheduling and allocating tasks to each of them. Level 3 is responsible for manipulating and visiting the knowledge base (KB). When a percept is received by an agent, Level 3 will automatically update the knowledge base. On the other hand when being asked about the current state, it will retrieve specific information from the KB, so we call it *reasoning level*. Level 4 *(physical level)* contains various physical implementations, including KB, network communication (TCP/IP), and special algorithms such as string processing, Dijkstra algorithm [4], breadth-first search algorithm [3], minimum cost flow algorithm [8] and Hungarian algorithm [5,7].

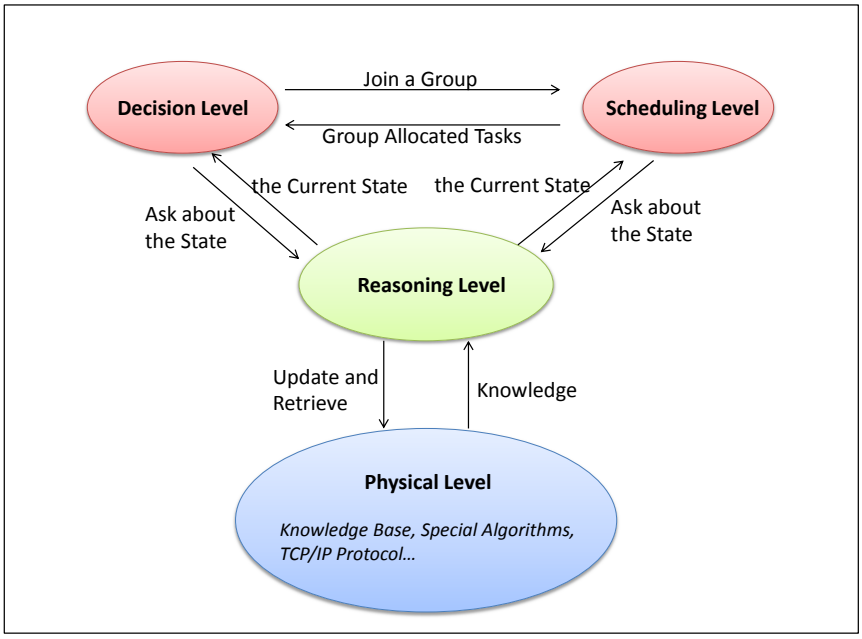

Fig. 1. Agent Model Diagram

To develop my system, I used Gedit Text Editor in Linux system, together with the g++ compiler. With the flexible C++ programming language, I was able to implement all the features of my system quite efficiently, so no features are lost in my implementation. Although I did not distribute the agents on different machines when I participated in the contest, I am actually able to do so with minor adjustments, that is, to modify the number of user names and passwords in the initialization file. When agents are on the same machine, they communicate with each other by sharing main memory, otherwise they do so via the TCP/IP protocol. In the receive-percept period, if an agent receives a new percept, she will immediately perform reasoning to figure out her current state and update her knowledge base. In the meantime any other agent will neither think nor perform actions, until this update is completed. In the send-action period, each agent reasons on her knowledge base to figure out her state, then reacts according to our previously computed classification, before the action is sent to the server. Furthermore a multi-thread TCP/IP sender will send the action to the server. Note that our program is so efficient that any agent is always able to send her action to the server before the next percept arrives. The most difficult part of the whole development process was the optimization of team strategies. That is, how to classify all the possible states and how to compute the optimal action wrt each specific class. Roughly I solved these problems after a series of observations, experiments, and comparisons. In classification, I considered roles, injury, emergency, etc, and in the end, there were nearly 100 specific classes. For example, suppose there is an agent who knows that her role is a repairer and that she is neither injured nor in emergency, e.g., her energy value is too low. And if there is an injured teammate in her location, she will retrieve all these pieces of information from her knowledge base and consider all these factors to compute which specific state she is in. And she will finally generate a reaction to repair the injured. To design agents who react responsively and effectively, I classified all possible states, and for each class, compute the optimal response beforehand. In total, I wrote 10,000 lines of C++ code for my system.

4 Strategies, Details and Statistics

The main strategy of my agent system is that the whole team survey the edges and probe the nodes of the whole map to search for available areas, and then they try to occupy areas with higher values. If any minor event occurs, such as encountering enemies or getting injured, the agent will abort her task. No matter whether they are exploring a map or trying to occupy some area, the agents will cooperate in an optimal manner, avoiding redundant work, so that they are able to accomplish the task with the lowest cost.

4.1 Task Allocation

Given a set of tasks $w[1, \ldots, n]$ and the same number of agents $a[1, \ldots, n]$, an arrangement can be denoted as a matrix $Ar_{n \times n}$, where $Ar_{i,j} = 1$ if task w_i is

allocated to agent a_j, otherwise, $Ar_{i,j} = 0$. Here our strategy is that each of the agents is allocated exactly one task, so in each of the rows and columns of Ar, there is exactly one '1'. We use matrix $C_{n\times n}$ to denote the costs (the number of steps or energy value an agent needs to accomplish a task), where $C_{i,j}$ denotes the cost needed for agent a_i to complete task w_j. Considering all possible arrangements, we hope to find a minimal value v such that each agent accomplishes her allocated task with costs no more than v. Let S be the set of possible arrangements such that the maximum cost is minimal, and let T be the elements in S such that the total cost is minimal. Algorithm 1, as shown in the following, returns one element in T. It involves two procedures, `Maximum_matching(Agents,Tasks,Edges)` based on Hungarian algorithm [5,7], and `Min_cost_flow(source, sink, Agents, Tasks, Edges, Cost)` which is just the one in [8]. Table 1 shows the test results of Algorithm 1. Each row shows a specific type of 10 experiments, where the first three columns show the number of agents, tasks and edges respectively. The fourth here shows the average number of edges whose value is not greater than v . The last column shows the average running time.

We allocate each agent a unique task so that repetitive work is avoided so that we are able to minimize the total cost. As mentioned earlier, when any agent receives a new percept, any other agent will not perform any actions until this percept is passed to all of them. This ensures that all agents share a synchronized knowledge base based on the presumption of common knowledge. Each time an agent arrives at an unexplored location, she surveys this location, obtaining all adjacent nodes and the costs of respective edges. In this way, all locations explored form a connected component and the agents know all information about this subgraph, including the shortest path between any two nodes in this component. Their strategy now is to move to those nodes on the boundary, survey them and then continue this process again and again. This will accelerate the process of searching for more valuable areas. To avoid the case that two agents move to the same location to survey, and to minimize the total cost, we use Algorithm 1 to inform each agent where they should go. To communicate with the server, we use a multi-threaded TCP/IP protocol.

I have designed a particular strategy for each of the five roles in the game. When an agent realizes that she is acting in a certain role, say, repairer, she will follow the respective strategy. Only explorers will accept the mission of exploring the map and probing the value of the newly encountered node. After finishing exploring, they will join a group to conquer a large zone. Here sentinels will join a group to survey all the edges and after that, will join another group to conquer a large zone just as what the explorers do. If some enemies are found and the team does not know their roles and specific states, inspectors will join a group to inspect those enemies, to collect such information. Otherwise, they will join a group to survey all the edges and then join another group to conquer a large zone just as what sentinels do. If some injured teammates are found, repairers will run to them and repair them, otherwise, they will join a group to survey the edges and then another group to conquer a large zone in the same way.

```
input : Agents, Tasks, Cost
output: arrangements
// binary search
low ← min_{x∈Agents,y∈Tasks} Cost(x,y) − 1
high ← max_{x∈agents,y∈Tasks} Cost(x,y)
while low+1 ≤ high do
    mid ← ⌊(low+high)/2⌋
    Edges ← {(x,y)|x ∈ Agents, y ∈ Tasks, Cost(x,y) ≤ mid}
    if Maximum_matching(Agents,Tasks,Edges) == |Tasks| then
    // Hungarian algorithm
        high ← mid
    else
        low ← mid
    end
end
Edge ← {(x,y)|x ∈ Agents, y ∈ Tasks, Cost(x,y) ≤ high}
Edges ← Edges ∪ {(source,x)|x ∈ Agents} ∪ {(x,sink)|x ∈ Tasks}
Cost(source,x) ← 0// for all x ∈ Agents
Cost(x,sink) ← 0// for all x ∈ Tasks
Min_cost_flow(source, sink, Agents, Tasks, Edges, Cost)
for (x,y) ∈ Edges do
    if flow(x,y)==1 then
        // flow is defined in Min_cost_flow
        arrangements(x) ← y
    end
end
return arrangements
```

Algorithm 1. Min_max_cost_tast_allocation(agents,works,cost)

Table 1. Experimental results for Algorithm 1 (value of edge < 10000)

Agents	Tasks	Edges	Remaining edges	Time
20	300	3000	62	0.0039s
20	1000	10000	65	0.0088s
100	1000	50000	485	0.0571s
200	1000	100000	1243	0.1634s
500	1000	250000	3641	0.8317s
1000	1000	500000	8153	4.5360s

If some enemies are discovered, saboteurs will go to front line and fight with those enemies, otherwise, they just do what sentinels do in the same occasion.

4.2 Expanding Zones

Expanding a boundary node B, means adding all adjacent nodes of B, which are not occupied by the enemies, into the current zone. The agents first choose

```
input : Nodes, Edges, value, Enemy_Nodes
output: Best_Zone
for x ∈ Nodes do
    Neighbor_x ← {y|(x,y) ∈ Edges}
end
Can_Not_Expand ← {x|x ∈ Enemy_Nodes or Enemy_Nodes ∩ Neighbor_x ≠ ∅}
// cannot be Expanded if an enemy is at or right beside
for i = 0 to p_2 − 1 do
    // p_2 is a prime number, assumed 1000007
    Hash_Zones_i ← ∅
end
for start_node ∈ Nodes do
    Bound ← {start_node}
    Zone ← Bound
    while ∃x.x ∈ Bound ∧ (Neighbor_x − Zone − Enemy_Nodes ≠ ∅) do
        // there exists a non-enemy point right outside the boundary
        if Bound ⊆ Can_Not_Expand then // no point can be Expanded
            S ← {x| min_{x∈Bound} |Eat_Nodes_x|}
            // the set of points needing the least agents if eating
            T ← {x| max_{x∈S} Σ_{y∈Neighbor_x−Zone−Enemy_Nodes} value_y}
            // set of nodes in S maximizing total cost
            Zone ← Zone ∪ {x|x ∈ Neighbor_y − Zone − Enemy_Nodes ∧ min_{y∈T} y}
            // Select any point in T, expand it
        else
            S ← {x| min_{x∈Bound−Can_Not_Expand} |Expand_Nodes_x|}
            T ← {x| max_{x∈S} Σ_{y∈Neighbor_x−Zone} value_y}
            Zone ← Zone ∪ {x|x ∈ Neighbor_y − Zone ∧ min_{y∈T} y}
            // Select any point in T, expand it
        end
        Bound ← {x ∈ Zone|Neighbor_x ⊈ Zone}
        if Σ_{x∈Best_Zone} value_x < Σ_{x∈Zone} value_x then
            Best_Zone ← Zone
        end
        hash ← (Σ_{x_i∈Zone,0≤i<|Zone|} x_i × p_{1i}) mod p_2
        // p_1 and p_2 are prime numbers, p_1 is 1007 and p_2 is 1000007
        if Zone ∈ Hash_Zones_hash then
            break
        end
        Hash_Zones_hash ← Hash_Zones_hash ∪ Zone
    end
end
return Best_Zone
```

Algorithm 2. Expand(Nodes, Edges, value, Enemy_Nodes)

Table 2. Experimental results for Algorithm 2

Nodes	Edges	Enemies	Time	Time (distributed)
100	300	20	0.0207s	0.002s
200	600	20	0.2910s	0.006s
300	900	20	1.2581s	0.013s
400	1200	20	3.5527s	0.023s
500	1500	20	7.4012s	0.034s
1000	3000	20	> 30s	0.128s

each node which is not occupied by enemies as a point zone and then repeat the following: find the boundary node P such that after expanding P the boundary increases the least (possibly by a negative number), and then, expand it. During the expanding process, we maintain the best zone found in the past, with the highest value. We say a zone B_1 is superior to another one B_2 if B_1 is more valuable than B_2. In details, we have the following Algorithm 2. The complexity of this algorithm is $O(N^2M)$, where N is the number of nodes and M is the number of edges in the graph. This is because the zone will only be expanded at most N times and at each expanding, at most M edges will be traversed. Table 2 shows the test results of Algorithm 2, where the first two columns show the number of nodes and edges respectively, and the third column shows the number of enemies, that is, the number of nodes occupied by enemies. The last two columns show the average running time for centralized and distributed algorithms respectively. Notice that in each type of experiments, the sum of the running time over all the machines for the distributed algorithm, is several times greater than the running time of the centralized algorithm, because in the centralized algorithm, we apply the hashing technique to examine whether a zone had already been computed before.

Note that Algorithm 2 can be made distributed, in that the expanding procedure can simultaneously begin at any number of nodes on the map. In particular, if we have as many machines as the nodes, we allocate each machine a unique node and instruct it to run a separate expanding procedure with that node.

4.3 Strategy Details

Formally below is the evaluation function for estimating the value of a zone:

$$\texttt{value}_{\texttt{Zone}} = \sum_{\texttt{i} \in \texttt{Zone}} \texttt{value}_{\texttt{i}}. \tag{1}$$

Our agents will calculate the most promising zone with Algorithm 2 and then move to the boundary of that zone and conquer it. Among them, the saboteurs always attack the nearest agent of the rival, so that this group of agents always attack the nearest area occupied by the enemies. If they are attacked

by the enemies, they will recompute a new area not occupied by the enemies, and then move there. All agents are equipped with exactly the same program, however, at each step during the contest, the strategy can be changed with a relatively small probability. Intuitively given an area, the safest strategy is to fully cover its boundary, that is, each boundary node is occupied by an agent. However, we sometimes take some risk, hoping to occupy more with the same number of agents. One possible risky strategy is that there is at least an agent at any two adjacent boundary nodes. At the start of the contest, we exploit such risky strategy to conquer an area. If this area is often disturbed by enemies, we will recompute a new area with the aforementioned safe strategy and then move there. To summarize, two factors can trigger strategy changes: (1) whether a conquered area is often disturbed; (2) a relatively small probability. During the procedure of path finding, we exploit Dijkstra Algorithm and Breadth-First Search Algorithm, and we also use Algorithm 1 to prevent any two agents from exploring the same location. During the contest, there is a certain strategy that only saboteurs will buy sabotage device and shield, and the strength value will always be equal to the health value or one unit more. However, according to empirical results, it is best not to buy any facilities. Considering that this does not cause big problems, at the start we randomly make a choice between these strategies. In the contest, we value achievements, from which we are able to obtain some scores at each step, so we try to acquire achievements swiftly, never spending them.

As mentioned earlier, all agents in our team are rational and good team players, that is, each will always try to complete the mission of the group. Moreover, recall that all communications are perfect and all agents will not perform any actions when a certain percept is being passed in the group. In our team all the agents are armed with exactly the same program so that they have equal status. When a list of agents are applying for the same mission, one of them will become a temporary project manager, which is responsible for allocating the mission in an optimal way. Later this project manager will become an ordinary agent and each agent will accomplish her allocated mission separately. Hence we organize our agents explicitly and no hierarchy is exploited. When an agent encounters something emergent, she immediately interrupts her allocated mission and tell all others in the group. The group will possibly relax the team mission so that they are able to accomplish it without this agent. Agents are able to perform planning in path finding and they need complete knowledge about the (local) initial state. Here we do not call a planner, but exploit Dijkstra Algorithm to obtain a shortest path from the source to the destination. To synchronize with the server, the agents use multi-thread TCP/IP listeners to listen to the message from the server, and decide what actions to perform accordingly. Furthermore, a multi-thread TCP/IP sender will send the action to the server. Note that our program is so efficient that any agent is always able to send her action to the server before the next percept arrives.

5 Conclusion

The participation of this contest has greatly improved my knowledge of multi-agent systems and stimulated my interest in conducting research in this area. I have learnt some important strategies to improve the performance of my agent team. Firstly, agents should be trained beforehand to strengthen the preconditions of their actions in order to reduce the search space. For example, the agents would realize that any node should not be surveyed repeatedly so they strengthen the precondition of the survey action. Secondly, the agents should record some optimal solutions in some cases, then with the learned experiences, they will be able to make best responses in similar cases. For instance, if a saboteur encounters an enemy for the first time, she deliberates over the optimal strategy, attacks that enemy, and learns that experience. Then if similar cases happen next time, she will simply behave according to this experience, without deliberation. Thirdly, the agents should keep a balance between maximizing their worst outcome and minimizing the best outcome of their enemies in the meantime.

One strong point of our team is that we use Algorithm 1 for task allocation to avoid repetitive work, hence decreasing cost of the team. Also, Algorithm 2 ensures that our agents are able to search for a large area, and then occupy it. Another is that our team is efficient in that it only takes the team about 0.2 second to make all decisions, on the 300-edge and 800-node map, in a perfect network. This enables us to develop more complex strategies in future contests. The weaknesses of our team are that we do not have a good strategy for disturbing the opponents and we are not able to defend our own area effectively. Because there is a great number of agents and the map is complex, our programs have to run with great efficiency. Hence we choose C++, which is known for its efficiency and flexibility, supporting various data structures and algorithms. Next year we are going to exploit effective strategies to attack enemies' zone and protect our own zone. The performance this year is not so satisfactory and there are many reasons: this was the first time for us to participate, the team consisted of only one member, I have just finished my undergraduate study with little research experience, and I had not enough time to implement all the ideas. For the next year, some changes we would think beneficial include: (1) servers should never send repetitive static information so as to relieve the pressure of network communication; (2) a percept should contain no information about the teammates because the agents should communicate with each other to broadcast such information.

Acknowledgements. I thank Professor Yongmei Liu for introducing me to the Multi-Agent Programming Contest. I am deeply grateful to Yi Fan for his generous and valuable help with the writing of this paper. This project was supported by the Natural Science Foundation of China under Grant No. 61073053.

References

1. Behrens, T., Dastani, M., Dix, J., Köster, M., Novák, P.: Special issue about Multi-Agent-Contest I. In: Annals of Mathematics and Artificial Intelligence, vol. 59. Springer, Netherlands (2010)
2. Behrens, T., Dix, J., Köster, M., Hübner, J.: Special issue about Multi-Agent-Contest II. In: Annals of Mathematics and Artificial Intelligence, vol. 61. Springer, Netherlands (2011)
3. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Section 22.2: Breadth-first search. In: Introduction to Algorithms, pp. 531–539. MIT Press and McGraw-Hill (2001)
4. Dijkstra, E.W.: A note on two problems in connexion with graphs. In: Numerische Mathematik, vol. 1, pp. 260–271. Springer (1959)
5. Edmonds, J.: Maximum matching and a polyhedron with $0,1$ vertices. J. of Res. the Nat. Bureau of Standards 69 B, 125–130 (1965)
6. Fagin, R., Halpern, J.Y., Moses, Y., Vardi, M.Y.: Reasoning about Knowledge. The MIT Press, Cambridge (1995)
7. Kuhn, H.W., Yaw, B.: The Hungarian method for the assignment problem. Naval Res. Logist. Quart, 83–97 (1955)
8. Orlin, J.B.: A faster strongly polynomial minimum cost flow algorithm. Operations Research 41(2), 338–350 (1993)