# Multiagent Parallel Distributed A*

**Tom Eliassy and Jeff Rosenschein**
Benin School of Computer Science and Engineering,
The Hebrew University Of Jerusalem, Israel

## Abstract

The Search technique makes it possible to solve a lot of our day-to-day problems in general and A* is the most appropriate algorithm for finding the optimal solution in particular. Since there are many problems that can be modeled using the Multiagent format by their nature and considering the remarkable improvements in the hardware field, we would like to examine whether it is possible to make the original A* Search algorithm more efficient such that it will utilize the abilities of the hardware technology. In this paper we describe a modified version to the algorithm that proposed by Nissim and Brafman (2012) for systems that can't use signals as the way of communication among the agents. The modification overcomes this limitation by using a shared resource that enables the agents communication. The paper also presents the circumstances for using the CSP methodology rather than the A* Search one regarding planning problems.

## General Terms

Algorithm, Optimality, Hardware, Signals, Shared Resource, Heuristic, Communication

## Keywords

Search, Planning, A*, Multiagent (MA), Constraint Satisfaction Problem (CSP), Loosely- Coupled, Branching Factor

## Introduction

Most of our daily activities consist of Search and Planning problems features that can be found in the fields of Transportation, Logistic, Industry, Science etc. These features, such as optimality, constraints, prioritization and cooperation between some parties can be a part of common activities like arriving at a destination from an origin via a road that we shared with other people. We may use it in several ways driving a car, train, motorcycle, bicycle, walking, etc. with limitations such as time and fuel, we are interesting in the most optimal route which enables the fastest option. Another example could be a product producing that consists of several stages that are done by different machines, tens workers and regarding a number of conditions.

Thanks to technological progressions and developments, we notice the significant achievements in the hardware field. We can also mention that according to the trend, the number of transistors in a dense integrated circuit doubles about every two years (Moore's Law; Moore and others (1965)). The latest improvements enable us to store more data on memory chips, to enable the calculation of several programs in parallel using enhanced computing power that wasn't possible before, to support the modern concepts such as Neural Networks and Graphic Processing (by GPU components). In this paper, we would like to examine the influence of these issues on the discipline of Artificial Intelligence, in general, and on the operation of A* search in particular.

At the same time that these methodologies evolved, there were breakthroughs in the software field focusing on Artificial Intelligence techniques. Among them we can note the revolutionary Deep Learning that tends to influence a variety of aspects in our lives, the Reinforcement Learning that is based on Markov model and puts itself in the front of unsupervised learning, Genetic Algorithms that borrow the name, as well as, the meaning from Biology and Logic that by performing reductions enables us to convert a lot of common problems to arranged formulation such that finding a solution for the translated clauses would lead us to solve the original problems that are taken from our day-to-day situations.

In addition to the last mentioned concepts, Search is one of the most fundamental methods for problem-solving and A* is probably the most qualified tool as heuristic Search algorithm that is known so far according to Nissim and Brafman (2012). We will use A* in the context of planning i.e by executing A* Search we are looking to find out the best solution (defines routes or order of operations) for a given planning problem. Furthermore, we are going to focus on MA planning problems that include multiple cooperative agents that have to achieve a common goal by integrating the agents' individual calculations using some aspects of communication between them. These problems have a nature and a structure that make it possible to divide each problem into sub-problems such that from a certain combination of the solutions for the sub-problems we may deduce how to solve the entire original problem. We would like to check whether the utilization of the mentioned hardware abilities by the MA modelling can reduce the overall computing time regarding to a classical centralized planner despite the overhead that is required for synchronization. We would also like to compare the A* Search to another approach for tackling MA planning problems, Constraint Satisfaction Problem (CSP).

We would like to propose a modified version to the algorithm of Nissim and Brafman (2012) that include agents communication using shared resource (i.e data structure) rather than signals. The modification can be applied in systems which don't support the use of signals (for example Python custom threads) or in cases that the frequency of the signals prevents from the agents to perform their essential individual calculations. As well as Nissim and Brafman (2012) algorithm, our version doesn't consider the time dimension and is also free of the loosely-coupling assumption. Despite this fact, it is possible that in some of such cases (loosely-coupled agents) there are optimal designated algorithms for solving them but it depends on the nature and structure of the specific problem. Additionally, our algorithm treats each agent as a black box: it doesn't interfere in an agent's internal planning, but transfers the result of this planning to the other relevant agents using distinguishing between public and private actions. With this manner, the algorithm can be fully distributed as each agent maintains its own private information or, in contrast, the agents can share their information with each other.

Additionally, our version provides a few more features that are a part of the technique's nature. In case we have only a single agent, the algorithm will actually produce the same result as the original A* Search would do. The optimality of the algorithm is directly derived from the optimality of A* Search algorithm and there is an elimination of redundant calculations for symmetric cases (for example, if agent 2 has to compute his planning on a state contains the planning of agent 1 and then send it to agent 3, it may be symmetric to the case where agent 1 has to compute his planning on a state contains the planning of agent 2 and then send it to agent 3 - in case that their planning are not intersected). Moreover, since we deal with a version of A* Search there is a need to reduce the branching factor as much as we can with respect to the specific problems discussed. Our modification enables each agent to use its own heuristic function. Furthermore, there is continuous execution of re-planning, until a goal state is found.

## Related Work

Regarding MA planning problems, there are two widespread approaches such that the first is based on CSP modelling and the second use a tree-structured model. Three out of four (Nissim and Brafman (2012), Brafman and Domshlak (2008), Nissim, Brafman, and Domshlak (2010)) related works distinguish between the agents public actions to the internal ones and the same rate of the related works: (Brafman and Domshlak (2008), Nissim, Brafman, and Domshlak (2010), Sharon et al. (2012), and Boyarski et al. (2015)) consider the time aspect, meaning that "wait" operations are taken into account. In other words, the agent who performs a "wait" action is actually suspended until some preconditions would be satisfied in order to execute an action which requires them. Additionally, half of the previous works are based on the centralized concept: Brafman and Domshlak (2008), Sharon et al. (2012), Boyarski et al. (2015) as opposed to the others that are distributed and use signals (Nissim and Brafman (2012)) or formulated variables clauses (Nissim, Brafman, and Domshlak (2010)) for communicating between the agents.

There are some unique characteristics of the CSP techniques that mentioned in Brafman and Domshlak (2008), Nissim, Brafman, and Domshlak (2010) previous works. Among them, is an assumption that the agents are loosely-coupled, meaning that we assume that there is a limited interaction between the agents' sub-plans. Consequently, there are problems that cannot be resolved in this manner due to high level of coordination that is needed in the case of tightly- coupled agents. Therefore, the related papers define parameters for measuring the problem's properties regarding some aspects such as problem independent and problem specific which indicate a problem's level of coordination Brafman and Domshlak (2008) and such the $\delta$ parameter which is the upper bound on the number of public actions per agent that determines the dimensions of the actions, times and requirements variable domains Nissim, Brafman, and Domshlak (2010). In the same way, these methodologies (Brafman and Domshlak (2008), Nissim, Brafman, and Domshlak (2010)) also use the agent interaction directed graph which is composed of vertices that represent the agents and edges that connect a pair of vertices (agents) for pointing out that at least one action belongs to one of the agents affects the functionality of the other agent. This graph plays an important role in determining how loosely-coupled a problem is and is also tied to the worst-case time complexity of planning for a MA system as was demonstrated in Brafman and Domshlak (2008) article. Finally, we would note that although the CBS and ICBS techniques described by Sharon et al. (2012) and Boyarski et al. (2015) respectively are designated to the MA Path-Finding problem, they can be adapted for other problem types.

### Flow Free

We'll use the platform of the Flow Free game in order to measure the performance of the MA parallel distributed A* algorithm. Flow Free is a puzzle game for Android and iOS released by American studio Big Duck Games in June 2012. The game takes place on a grid containing pairs of colored dots, such that, the uncolored squares are considered to be empty as its initial state, like is shown in figure 1.

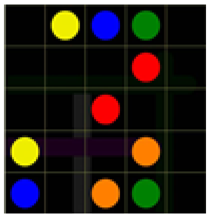

Figure 1: An example of an initial state of the Flow Free game

The objective is to draw flows (paths) between every pair of dots with the same color such that:

1. All the couples of dots will be connected

2. All the empty squares will be filled by the flows

Figure 2 contains an example of a Flow Free goal state.

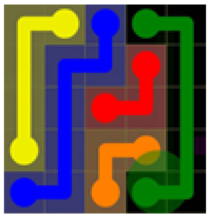

Figure 2: An example of a Flow Free goal state

The difficulty level is directly affected by the square size of the grid, ranges from 5X5 to 14X14 cells and there are also some expansions to the game:"Flow Free: Bridges" (November 2012) which contains bridges that have to be occupied by a flow that will pass it above other flow (may be the same flow), "Flow Free: Hexes" (October 2016)- Hexagon- shaped grids and "Flow Free: Warps" (August 2017)- Different squares- shaped grids with a connection between the board's edges [1].

We are looking for an algorithm that can deal with tightly-coupled agents and makes it possible to calculate the routes of the agents by parallel and distributed manner. Regarding the features of the Flow Free game, we would identify every color of flow with an agent. We will try to reduce the total overhead as much as we can without damaging the agents' interaction. Moreover, since an occupied square in the grid is immutable, there is no meaning to "wait" operations (because then no one can change the square to be vacant or contain a flow of other color) therefore we would like our algorithm to ignore the aspects of the time dimension. We would also note that the optimality of the solution is guaranteed because of the nature and definition of A* Search that is used. However, in the Flow Free game there is a need for finding a solution without considering its optimality.

## Multiagent Parallel Distributed A*

Multiagent (MA) A* is a distributed variation of A* that can be done in parallel calculation. We describe an algorithm which is an adaptation to the work of Nissim and Brafman (2012) for cases that signals-based communication between the agents is not possible, this is the contribution of this paper. The adaptation is also free of the loosely-coupled agents assumption and contains a shared-resource which is actually a data-structure that stores information. This information is necessary for the calculations of all the participant agents, so they have to communicate each other by using this data structure (in the fully-distributed case, this is the only data that is shared among the agents). Because of the shared-resource existence, there is a need to use a mutual-exclusion object that will limit simultaneous access to this shared- resource only for a single agent every time. In turn, every agent will store states that are relevant for the other agents (with specifying to which ones of them) and will also pick a state that an other agent has posted for him.

[1]https://en.wikipedia.org/wiki/Flow_Free

Since each agent has a different search space, in order to execute the algorithm, every agent has to contain some individual data structures. First, like in Nissim and Brafman (2012) essay, an agent has to have an internal open list which is actually a minimum priority- queue that stores the agent's local states that are candidates for expansion. In every iteration of the MA-A* algorithm, a state s with the minimum $f(s) = g(s) + h(s)$ value will be expanded. Apart from the open list, there is also a closed list which holds the already expanded states. In addition to Nissim and Brafman (2012), we would like every agent to maintain another priority- queue that will store the states which were posted by other agents as relevant for the current agent, we would denote it as external open list. The expansion prioritization will be as follows: first, an agent will try to expand a state in the external open list, which was taken from the shared-resource. If there is no such a state, the agent will try to expand a state from the internal open list and if that it is empty, the agent will go to sleep. A skeletal structure of an agent is demonstrated in the UML component of figure 3.

**Class Agent:**

**Attributes/Fields:**

- internalOpenList: PriorityQueue
- externalOpenList: PriorityQueue
- closedList:[]

**Functions/Methods:**

+ multiagent_astar (): void
+ expand (state): void
+ find_successors (state):[]

+ Post_state_for_other_agents ():void

Figure 3: A skeletal Agent class

The API of the MA-A* is composed of 4 integrated parts: general method, expansion and successors finding which are the fundamental elements of the A* Search algorithm as well as communication between the agents that is done by posting data to the global shared- resource in our case. The following algorithms (1 - 3) extend their corresponding versions in Nissim and Brafman (2012) by treating the management of the access to the shared resource by the mutual-exclusion object. Furthermore, we will distinguish between achieving a local (internal) goal state to achieving a global (external) goal state. While the first one means that the agent will post his local (internal) goal state to the agents who haven't completed their task using the shared-resource, according to the second an agent has to communicate with the other agents in order to notify them about a global (external) goal state that was found. There are few ways to fulfill it regarding to the level of information hiding: for instance, an agent can maintain a boolean flag or the program can use a global variable in order to indicate such an occasion.

## General algorithms of the MA-A* API

**Algorithm 1:** Multi-Agent A*

```
1. First expansion (The shared-resource and the openList
   are empty, no states to get)
2. while Not (global goal state)
   (a) Acquire the mutual-exclusion object
       i. if (there is a relevant state in the shared-resource
          which hasn't been expanded by this agent yet) then:
            • Get the state from the shared resource and
              expand it
   (b) Release the mutual-exclusion object
   (c) if (didn't get a state from the shared-resource) then:
       i. if (the openList is Not empty) then:
            • Get a state from the openList and expand it
       ii. else:
            • Go to sleep
```

**Algorithm 2:** Expand (state)

```
1. Insert state to the closedList (Mark as visited)
2. if (state is a global goal state) then:
   (a) return
3. successors $\leftarrow$ Find successors (state)
4. for Each successor that is NOT in the closedList:
   (a) Insert the successor to the openList
5. if (one of the successors is a local goal state) then:
   (a) Post the local goal state to the relevant agents using
       the shared resource
```

**Algorithm 3:** Find successors (state)

```
1. optional moves $\leftarrow$ Calculate the optional moves for the
   current agent
2. while (There is only a single optional move) do:
   (a) Fast-forwarding (expand the successor of the single
       optional move)
3. for every move in Optional moves:
   (a) if (The move has to be eliminated) then:
       i. add the move to the closedList
4. return Optional moves
```

## Experimental Results

In order to match the proposed algorithm to the Flow Free platform we performed some adaptations. First, as the shared resource we used a map that stores for every agent the relevant states that the other agents have posted for him, the order is determined according to these state's prioritization. To avoid multiple simultaneous accesses to the mentioned data structure, we chose a semaphore as the mutual exclusion object since it is free of deadlock and starvation. Secondly, we defined the heuristic function $h$ for a state to be the number of empty squares in its board, one should pay attention that this is not an admissible heuristic function. Besides it, for an agent, the cost function $g$ of a state returns the number of moves that the agent has performed on the state's board so far, without taking into account forced moves[2]. Moreover, while transferring a state for an agent via the shared resource it is important to zero the cost value of the state in order that the recipient agent will prioritize the transferred state over his own states. Most of the time this is what actually happens since the prioritization of state s is calculated according to the sum function $g(s)+h(s) = f(s)$. When the value $g(s)$ is minimal and equals to 0, $f(s)$ is smaller than almost every $f(t)$ value as t is a successor of the initial state that has been processed only by the recipient agent (and therefore $f(t) \approx f(init.state)$). In this way, an agent would prefer states that were sent from the other agents over the states which were computed only by himself. With the same principle, an agent would usually prioritize the state contains the highest number of completed flows that were done by the other agents.

From looking at table 1 which contains the results of the measurements one can infer some understanding about the behavior of the MA-A*. Beginning with the fact that because of the parallel computing aspects, the values (time and number of expanded) of the MA-A* performance are treated as an average[3]. One can also notice that there is nodes-explosion when executing MA-A* comparing to the traditional A* Search. Meaning, despite the fact that the MA-A* allocates more hardware resources in favor of the calculations, there are a lot of redundant ones which increase the running-time, although the trivial known fact that the C programming language much faster than Python. Moreover, comparing to the CSP solver, it can be discerned that there is an asymptotic difference from the run time of MA-A*. This difference grows as the size of the grid is increased and leads the run time of MA-A* to be exponentially in the run time of the CSP solver.

[2]A move is considered to be forced if a flow has only one single possible square to move into or an empty square is adjacent to only one single flow's head and it has only one free neighbor

[3]The average values of MA-A* were taken regarding the mean of 5 observations.

| | MA-A* | | CSP | | | A* (C Implement) | |
|---|---|---|---|---|---|---|---|
| | **avg. Time** | **avg. Nodes** | **Time** | **SAT vars.** | **clauses** | **Time** | **Nodes** |
| **regular 5x5** | 0.23 | 102 | 0.003 | 184 | 2,031 | 0.001 | 16 |
| **regular 6x6** | 0.46 | 119 | 0.004 | 322 | 4,353 | 0.001 | 25 |
| **regular 7x7** | 3.01 | 1,060 | 0.007 | 452 | 6,418 | 0.001 | 40 |
| **regular 8x8** | 1.56 | 282 | 0.010 | 616 | 9,452 | 0.001 | 55 |
| **regular 9x9** | 9.82 | 1,607 | 0.021 | 1,009 | 17,595 | 0.001 | 166 |
| **extreme 8x8** | 31.36 | 5,769 | 0.011 | 571 | 8,472 | 0.001 | 76 |
| **extreme 9x9** | larger than 3 minutes | - | 0.017 | 661 | 9,112 | 0.001 | 111 |
| **extreme 10x10** | larger than 3 minutes | - | 0.019 | 938 | 14,818 | 0.034 | 4,625 |

Table 1: The measured results Using Intel(R) Core(TM) i5-5200 CPU @ 2.20 GHz 2.19GHz processor, the times are in seconds

## Conclusion

When we examine the time and space complexities of the MA-A* algorithm with regard to the Flow Free game, we can find that for a grid contains n squares the time complexity as well as the space one is O($n*3^n$) [4]. These complexities are composed of the complexities of A* and UCS [5] Search with branching factor of 3 (since in every move, the flow may be extended to one of 3 adjacent squares at most, except for a few special start states for an agent which enable the flow extension to one of 4 adjacent squares) multiple by n. This multiplication is caused by the checks [6] we perform to detect options for pruning as we generate the successors of a state. Finding such pruning options can reduce the branching factor of the search tree that may lead to drastically reducing of the run time, what we always aim for, especially when dealing with search trees. Relating to the Flow Free platform, one can notice that the run time complexity is affected only by the size of the grid such that an addition of colors to a given puzzle increases the run time complexity only by a polynomial rate as opposed to grid size increasing which results in exponential growth of the time complexity.

In addition to the last analysis, one may pay attention that since the CSP solver generates O(n) variables, the worst case run time of this technique is therefore O($2^n$). This finding leads us to an essential and fundamental conclusion: In case that the branching factor of our problem is greater than 2 and the number of CSP variables asymptotically equals to the max depth of the search tree, the CSP technique is more effective than using an algorithm based on search tree, like A* in general or MA-A* in particular.

Furthermore, when using the MA-A* algorithm, one has to pay attention to avoid duplicate calculations that are caused as a result of different running order in some cases. For example, if the planning of agents 1 and 2 don't intersect each other then as agent 3 gets a state from agent 2 after agent 1 completed his plan, this is the same state like agent 3 gets a state from agent 1 after agent 2 completed his plan. For this reason, while getting a state from the shared-resource that was posted by other agent, the recipient agent has to make sure that he had never got a similar state before. It is recommended to manage 2 closed lists: the first for internal expanded states and the second for states that were sent by the other agents, i.e external closed list. It may bring to a linear improvement of the run time.

Besides the above specific conclusions there are some more generic ones. In MA-A*, the Hardware (Processor, Cores) has a significant influence on the performance. Additionally, using parallelism doesn't necessarily reduce the running time (since there are issues of access to shared resources, overhead and maybe duplicated calculations). In addition, there are problems which have to be solved by a centralized manner due to their nature. Finally, a future work in this case may use graphplan heuristics for achieving more accurate heuristic function. On one hand, it can reduce the number of expanded nodes but on the other hand, the calculation for this type of heuristic functions increases the run time of the program, so it might be interesting to examine this trade-off.

[4]Since there is a finite and constant number of colors (agents), it doesn't have an influence on the mentioned complexities with respect to the O notation.

[5]Abbreviation of Uniform Cost Search

[6]The checks are detailed in `https://mzucker.github.io/2016/08/28/flow-solver.html`. Every check requires O(n) time and space.

## References

Boyarski, E.; Felner, A.; Stern, R.; Sharon, G.; Tolpin, D.; Betzalel, O.; and Shimony, E. 2015. Icbs: improved conflict-based search algorithm for multi-agent pathfinding. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*.

Brafman, R. I., and Domshlak, C. 2008. From one to many: Planning for loosely coupled multi-agent systems. In *ICAPS*, 28–35.

Moore, G. E., et al. 1965. Cramming more components onto integrated circuits.

Nissim, R., and Brafman, R. I. 2012. Multi-agent a* for parallel and distributed systems. In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems-Volume 3*, 1265–1266. International Foundation for Autonomous Agents and Multiagent Systems.

Nissim, R.; Brafman, R. I.; and Domshlak, C. 2010. A general, fully distributed multi-agent planning algorithm. In *Proceedings of the 9th International Conference on*

*Autonomous Agents and Multiagent Systems: volume 1-Volume 1*, 1323–1330. International Foundation for Autonomous Agents and Multiagent Systems.

Sharon, G.; Stern, R.; Felner, A.; and Sturtevant, N. R. 2012. Meta-agent conflict-based search for optimal multi-agent path finding. *SoCS* 1:39–40.