Multi-State Commitment Search

Yasuhiko Kitamura*, Makoto Yokoo**, Tomohisa Miyaji*, and Shoji Tatsumi*

*Faculty of Engineering, Osaka City University
3-3-138 Sugimoto, Sumiyoshi-ku, Osaka 558-8585, Japan
{kitamura,miya,tatsumi}@kdel.info.eng.osaka-cu.ac.jp

**NTT Communication Science Laboratories
2-4 Hikaridai, Seika-cho, Soraku-gun, Kyoto 619-0237, Japan
yokoo@cslab.kecl.ntt.jp

Abstract

We propose the Multi-State Commitment (MSC) method to speed-up heuristic search algorithms for semi-optimal solutions. The Real-Time A (RTA*) and the Weighted A* (WA*) are representative heuristic search algorithms for semi-optimal solutions and can be viewed as a single-state and an all-state commitment search algorithms respectively. In these algorithms, there is a tradeoff between the risk of making wrong choices in search process and the amount of memory for the recovery, with RTA* and WA* being the extremes. The MSC method introduces a moderate and flexible characteristic into these algorithms and can increase the performance dramatically in problems such as the N-puzzle. In this paper, by introducing a commitment list, we show a modification of RTA* and WA* to their MSC versions without violating their completeness. Then, we experiment with their performance in maze and N-puzzle problems, and discuss conditions that the MSC method is effective.*

1 Introduction

Problem solving in artificial intelligence can be formalized as a search process to find a solution path from the initial state to a goal state in a given state space graph. However, as the size of the state space graph increases, it becomes difficult for a simple search algorithm to find a solution with a practical amount of time and memory. To overcome this drawback, heuristic search algorithms such as the A* algorithm have been proposed that incorporate heuristic knowledge into their state evaluation functions [12].

Although the A* algorithm guarantees finding an optimal solution, it is more practical to find a semi-optimal solution in an allowable time period than to find an optimal solution taking more time. Such semi-optimal solution search algorithms include the Weighted A* (WA*) [12] and the Real-Time A* (RTA*) [8].

There is an apparent difference in the RTA* and WA* algorithms for finding a state to expand. WA* stores all the states in its search front as its expansion candidates, but RTA* reduces its candidates to only child-states of the previously expanded state. From a viewpoint of commitment in search process, RTA* is a single-state commitment (SSC) search because it limits the range of further search to descendants of a single state which has been just expanded. On the other hand, WA* is an all-state commitment (ASC) search since it does not limit the range at all.[1]

A semi-optimal solution search algorithm generally repeats a decision to choose a state to expand among one or more candidates, but a wrong choice may make the search to a goal state longer or infeasible. Hence, a SSC search like RTA*, which commits its further search to a single candidate, tends to degrade its performance because of making wrong choices. On the other hand, for easy recovery from wrong choices, it may be a good method to store all the candidates in the OPEN list as in WA*. However, this method can quickly consume memory as the search front spreads out and may actually make it infeasible to find a solution. Hence, there is a tradeoff between the risk of making wrong choices and the cost of memory for recovery, and SSC, such as RTA*, and ASC, such as WA*, are the extremes.

In this paper, we propose the Multi-State Commitment (MSC) method, that has a parameter n, which

[1] It can be viewed as a no-state commitment search because committing all states is equivalent to committing no state.

introduces a moderate and flexible characteristic to SSC and ASC. SSC and ASC are identical to MSC when $n = 1$ and $n = \infty$ respectively. MSC is flexible to balance the risk of making wrong choices and the cost of memory for recovery by changing n. In the following sections, we describe an incorporation of commitment list into the WA* and RTA* algorithms to modify them to their MSC versions without violating their completeness. Then, we apply these MSC search algorithms to maze and N-puzzle problems and show its remarkable effect on N-puzzle problems. Finally, we discuss conditions that MSC shows good performances and refer to related work.

2 Heuristic Search

A problem is given as a quadruplet $\langle S, O, s_0, G\rangle$ where $S(\neq \phi)$ is a set of states, O is a set of operators, $s_0(\in S)$ is the initial state, and $G(\subset S)$ is a set of goal states. The tuple $\langle S, O\rangle$ is called its state space graph. A child-state s' is generated by applying an operator $o = (s, s')$ to a state s. A state is said to be expanded when all of its child-states are generated.

A sequence of states obtained by successive operator applications is called a path, and a solution (path) is a path from the initial state to a goal state. If a search algorithm is able to find a solution when there is at least one, the algorithm is said to be complete. When a cost (> 0) is given to each operator, the solution cost is defined as the cost sum of the operators that construct the solution. A solution with cost c is optimal if there is no solution with cost less than c.

Heuristic search algorithms utilize a state evaluation function, or heuristic function, $h(s)$ that estimates the least distance from the state s to a goal state. When it does not overestimate the real distance, the heuristic function is said to be admissible.

3 Multi-State Commitment WA*

3.1 WA* algorithm

In the evaluation of a state s, the A* algorithm uses $f(s) = g(s)+h(s)$ where $g(s)$ is the estimated cost from the initial state to s, and WA* uses a weighted version of $f(s)$, namely $f(s) = (1 - W)g(s) + Wh(s)$ where $0 \leq W \leq 1$. Generally, as W increases, WA* finds a solution quicker but the solution quality worsens. In this paper, since we mainly have interest in the search speed for a semi-optimal solution, we deal with cases of $W = 1$ only in the following discussion.

Like A*, WA* uses two lists called an OPEN and a CLOSED lists. Generated states are stored once in the OPEN list and expanded states are moved into the CLOSED list, so it is guaranteed that generated states are in either the OPEN or CLOSED list. Hence, WA* is able to avoid generating an identical state twice by checking these lists.[2]

WA*
1: $s \leftarrow s_0$;
2: generate $successors(s)$;
3: **if** $successors(s) \cap G \neq \phi$ **then return** $success$;
4: $add(OPEN, successors(s))$;
5: $add(CLOSED, s)$;
6: **if** $OPEN = \phi$ **then return** $failure$;
7: $s \leftarrow get\_min(OPEN)$;
8: **goto** 2;

Figure 1: WA* algorithm.

We show a WA* algorithm in Figure 1. WA* begins with the initial state (Line 1). It expands a state and generates child-states, but it does not generate any states which are in OPEN or CLOSED (Line 2). If there is a goal state in generated states, then it ends with success (Line 3). Otherwise, it adds generated states to OPEN (Line 4) and the expanded state to CLOSED (Line 5). If OPEN is empty, then the search ends with failure (Line 6). Otherwise, it chooses a state with the lowest h from OPEN (Line 7), and repeats the process (Line 8). Hereafter, these operations are counted as a single step.

3.2 MSC-WA* algorithm

WA* is an ASC search algorithm because it stores all the generated states in the OPEN list and chooses an expansion state from there. The Multi-State Commitment WA* (MSC-WA) limits the number of candidates to n, so we introduce a COMMITMENT list into WA* and modify it so that generated states are added to COMMITMENT, not OPEN, and an expansion state is chosen from COMMITMENT. MSC-WA* maintains the length of COMMITMENT at less than or equal to n. When the length is greater than n, it moves states to OPEN in a decreasing order from the highest h. If less than n, it moves states from OPEN in an increasing order from the lowest h while OPEN is not empty. Operations in CLOSED are the same as in WA*. Hence, even in MSC-WA*, it is guaranteed that generated states are in the COMMITMENT, OPEN,

[2] When we have interest in the solution quality, WA* should regenerate a state (move a state from the CLOSED list to the OPEN list) when its f is improved. In this paper, we have more interest in the search speed and a fixed $f(s)(= h(s))$, so we do not incorporate such operations into the algorithm.

MSC-WA*
1: $s \leftarrow s_0$;
2: generate $successors(s)$;
3: **if** $successors(s) \cap G \neq \phi$ **then return** $success$;
4: $add(COMMITMENT, successors(s))$;
5: $add(CLOSED, s)$;
6: **while** $length(COMMITMENT) > n$ **do**
7: $s \leftarrow get\_max(COMMITMENT)$;
8: $add(OPEN, s)$;
9: **while** $(length(COMMITMENT) < n)$ **and**
10: $(OPEN \neq \phi)$ **do**
11: $s \leftarrow get\_min(OPEN)$;
12: $add(COMMITMENT, s)$;
13: **if** $(COMMITMENT = \phi)$ **then return** $failure$;
14: $s \leftarrow get\_min(COMMITMENT)$;
15: **goto** 2;

Figure 2: MSC-WA* algorithm.

or CLOSED list. We show a MSC-WA* algorithm in Figure 2.

MSC-WA* does not generate child-states which are in either of COMMITMENT, OPEN, or CLOSED (Line 2), and adds generated states to COMMITMENT (Line 4). It adds the expanded state to CLOSED (Line 5). The length of COMMITMENT list is maintained at less than or equal to n (Line 6-12). If COMMITMENT is empty, then it ends with failure (Line 13). Otherwise, it chooses a state with the lowest h from COMMITMENT for the next expansion(Line 14). Ties are broken randomly.

MSC-WA* is equivalent to WA* when $n = \infty$.

3.3 Completeness of MSC-WA*

Here we show that MSC-WA* is complete. The completeness of WA* is guaranteed when there is at least a path from the initial state to a goal state and the state space graph is finite. This is because WA* does not fall into an infinite loop as it does not expand an identical state more than once, and because also it keeps at least a state on a solution path in the OPEN list before it terminates.

As with WA*, MSC-WA* terminates without falling into an infinite loop because it does not expand an identical state more than once. If it ends with failure, COMMITMENT must be empty (Line 13), and OPEN also must be empty because of the operations in Lines 9-12. However, since at least one state is on a solution path in COMMITMENT or OPEN, they cannot be simultaneously empty, and MSC-WA* never ends in failure. Hence, MSC-WA* is complete when there is at least a solution and the state space graph is finite.

RTA*
1: $s \leftarrow s_0$;
2: generate $successors(s)$;
3: **if** $successors(s) \cap G \neq \phi$ **then return** $success$;
4: $update\_h(s)$;
5: **if** $successors(s) = \phi$ **then return** $failure$;
6: $s \leftarrow get\_min(successors(s))$;
7: **goto** 2;

Figure 3: RTA* algorithm.

4 Multi-State Commitment RTA*

4.1 RTA* algorithm

RTA*, proposed by Korf [8], is a semi-optimal solution search algorithm which interleaves a look-ahead search and a move. By increasing the depth of the look-ahead search, the solution quality is improved. Since we mainly have interest in the search speed, we set the depth at 1 in the following discussion. RTA* do not use a CLOSED list like WA*, but it updates heuristic values during its search process to guarantee its completeness. We show the RTA* algorithm in Figure 3.

RTA* begins with the initial state (Line 1) and then generates all the child-states as expansion candidates. When the heuristic value of a child state is ∞, it is removed from the candidates (Line 2). If a goal state is included in the candidates, RTA* ends with success (Line 3). It will otherwise update the heuristic value of the expanded state s to second-best $c(s, s') + h(s')$ in those candidates where s' is one of the child-states of s. When there is only one or no child-state, the heuristic value is set to be ∞ because the expanded state is not on any solution path and does not need be expanded again (Line 4). If there is no following expansion candidate, RTA* ends with failure (Line 5). It will otherwise choose a state with the lowest h of the candidates and expand it in the next step. Ties are broken randomly (Lines 6 and 7).

It must be noted that there are two alternative interpretations of the RTA* algorithm. One interpretation is that this algorithm is a fast search algorithm that can produce a semi-optimal solution rapidly. Another interpretation is that this is an on-line algorithm such that an agent is interleaving planning and actions in the real-world. In this paper, we employ the former interpretation, and try to further improve the efficiency of the RTA* algorithm.

MSC-RTA*
1: $s \leftarrow s_0$;
2: generate $successors(s)$;
3: **if** $successors(s) \cap G \neq \phi$ **then return** $success$;
4: $update\_h(s)$;
5: $add(COMMITMENT, successors(s))$;
6: **while** $length(COMMITMENT) > n$ **do**
7: $\quad remove\_max(COMMITMENT)$;
8: **if** $COMMITMENT = \phi$ **then return** $failure$;
9: $s \leftarrow get\_min(COMMITMENT)$;
10: **goto** 2;

Figure 4: MSC-RTA* algorithm.

4.2 MSC-RTA* algorithm

RTA* is a SSC search algorithm that limits the candidates for the next expansion to descendants of a single state. Here we enlarge the range by modifying RTA* to MSC-RTA*. As with MSC-WA*, we incorporate a COMMITMENT list into RTA* and store some of generated states in the list as expansion candidates. The length of COMMITMENT is maintained at less than or equal to n. When it is greater than n, MSC-RTA* removes states from COMMITMENT in decreasing order from the highest h. We show a MSC-RTA* algorithm in Figure 4.

Lines 5 through 8 in Figure 4 of MSC-RTA* are substituted for line 5 and 6 in Figure 3 of RTA*. Namely, each of generated states is stored once in the COMMITMENT list if it is not in it (Line 5). The length of COMMITMENT is maintained at less than or equal to n. If it is greater than n, MSC-RTA* removes states from COMMITMENT in a decreasing order from the highest h (Lines 6 and 7). Then, if COMMITMENT is empty, it ends with failure (Line 8). It will otherwise choose a state with the lowest h from COMMITMENT for the next expansion. Ties are broken randomly (Line 9).

MSC-RTA* is equivalent to RTA* when $n = 1$.

4.3 Completeness of MSC-RTA*

RTA* is complete when, a goal state is reachable from all the states, the initial value of every h is finite, and the state space graph is finite [8].

We here show that the completeness of RTA* is preserved even if we modify it into MSC-RTA*. Because the initial value of every h is finite and there is at least a path from the initial state to a goal state, MSC-RTA* never fails with the infinite h value of every states surrounding the goal state.

MSC-RTA* never falls into an infinite loop because of the following reasons. If it fell into an infinite loop, a goal state could not be on the loop. The h value of an expanded state s is updated to second-best $h(s') + c(s, s')$ where s' is a child-state and $c(s, s')$ is greater than 0, so the updated h value is greater than the best h among its child-states. Hence, every trip of MSC-RTA* around the infinite loop increases the lowest h on the loop so the h values of all states on the loop increase without bound. At some point, since the h value of a state on the loop will be greater than that of a state not on the loop, the algorithm will escape from the loop. The number of states is finite, so it eventually reaches a goal state. MSC-RTA* is complete because this characteristic is preserved for every n.

5 Experimental Performance Analysis

To evaluate the performance of the MSC method, we executed experiments using maze and N-puzzle problems.

A maze is a 120×120 grid space where the entrance is located at (0,0) and the exit is located at (119,119), and its solution is a path from the entrance (initial state) to the exit (goal state). In the grid space, obstacles are located randomly at a ratio of 40%. There are four operations; moving UP, DOWN, RIGHT, and LEFT, with the cost of 1. The heuristic function of a state is given as the Manhattan distance from it to the goal state.

For the N-puzzle, we use a 48-puzzle with 48 numbered tiles arranged on a 7×7 board. The goal is to transform the given initial state to the goal state, where the tiles are sorted, by sliding tiles onto an empty square. There are four operators that move the empty square UP, DOWN, RIGHT, and LEFT, with the cost of 1. The heuristic function of a state is given as the Manhattan distance's sum of misplaced tiles.

We prepared 100 mazes with randomly generated obstacle patterns and with at least a solution, and 100 solvable puzzles with different initial patterns. Changing n, we executed 100 trials for each combination of MSC-RTA* or MSC-WA* and a problem. Since there is a memory bound for the computer used in the experiments, we aborted the algorithm when the number of states in all lists exceeded 1.5 million.

We show the obtained results of MSC-WA* in Table 1 (maze) and Table 2 (puzzle) and those of MSC-RTA* in Table 3 (maze) and Table 4 (puzzle). The figure in parentheses after the algorithm name represents n. The Success Rate is the percentage of suc-

cess without abortion due to the memory bound. The Search Steps and the Solution Length are the averages of successful trials.

In maze problems, neither of MSC-WA* or MSC-RTA* show any remarkable improvement for any n. In puzzle problems, on the contrary, both algorithms showed remarkable improvements. In contrast to WA*, which solved only 10% of the problems due to the lack of memory, MSC-WA*, with $n = 4$, solved all the problems. It is interesting that $n = 4$ had the best result and more or less than that makes the performance worse. The performance of WA* looks well concerning the solution length, but this means WA* found only short solutions and failed with abortion for long ones.

In experiments with MSC-RTA*, RTA* could solve none of the problems, whereas, MSC-RTA*, with $n = 2$ through 6, solved all of the problems. The performance was best when $n = 3$, and it degraded when n was greater or less than 3.

Due to the length of this paper, we omitted the results of 24-, 35-, and 63-puzzles, however, the algorithms showed similar performance and we believe this dramatic performance improvement can be achieved in any N-puzzles.

6 Discussion

The experimental results in the previous section show the MSC method improves the search performance dramatically in N-puzzle problems. On the other hand, it does not improve the performance in maze problems. In the following subsections, we discuss about the following questions.

- Why is the MSC method effective at N-puzzles, but not at mazes?
- Why is the performance improvement at N-puzzles so devastating?

6.1 Wrong Choice and Recovery

A semi-optimal solution search algorithm repeats a choice of an expansion state from one or more candidates and a wrong choice makes the search process longer or infeasible. Since a SSC search algorithm like RTA* limits the candidates to a single state, a wrong choice may cause a fatal result. On the other hand, because an ASC search algorithm like WA* stores all of the candidates, it expands the search front widely and consumes too much memory space in finding a solution actually. A MSC search algorithm is somewhere in the middle of these two methods and is balanced between the risk of making wrong choices and the amount of memory for recovery by changing n, which is the maximum length of commitment list.

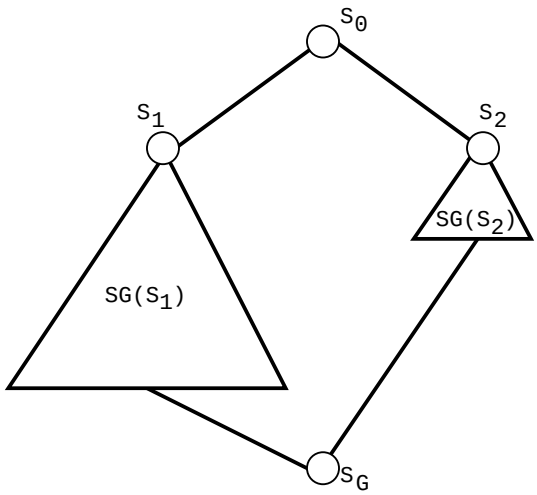

Figure 5: A search graph.

Here we explain the problems where MSC shows better performance than SSC by using a search graph shown in Figure 5. This search graph has an initial state s_0 with two child-states s_1 and s_2. Each of these child-states starts a partial graph search of $SG(s_1)$ and $SG(s_2)$ respectively. We assume the search in $SG(s_1)$ takes a long time to reach a goal state s_G while that in $SG(s_2)$ takes a short time. We also assume the heuristic function is misleading, namely $h(s_1) \leq h(s_2)$, and causes a wrong choice.

In this problem, SSC behaves as follows. It begins with s_0 and generates the child-states s_1 and s_2. Then, it chooses s_1 as the next state to expand due to the misleading h, proceeds to the search in $SG(s_1)$, and takes a long time to reach a goal state.

On the other hand, MSC behaves as follows. As with SSC, it begins with s_0, generates the child-states s_1 and s_2, and chooses s_1 as the next state to expand. However, it stores s_2 in the commitment list as another candidate. Hence, since $h(s_2)$ eventually becomes the minimum in the commitment list during the search in $SG(s_1)$, MSC aborts the search in $SG(s_1)$, restarts it from s_2, and reaches the goal state.

Hence, if MSC can abort the search in $SG(s_1)$, it shows a better performance than SSC. In other words, it is required that s_2 has been stored in the commitment list and $h(s_2)$ becomes lowest in it before terminating the search in $SG(s_1)$. Generally, a larger n raises the probability that s_2 is stored.

On the other hand, MSC may cause a side effect

Table 1: Performance of MSC-WA* in maze.

Algorithm	Success Rate	Search Steps	Solution Length
MSC-WA*(1)	100	1233.4 (728.0)	383.2 (58.3)
MSC-WA*(2)	100	1228.8 (717.0)	374.1 (55.3)
MSC-WA*(3)	100	1223.2 (708.4)	373.6 (55.9)
MSC-WA*(4)	100	1232.6 (712.3)	373.8 (55.5)
MSC-WA*(5)	100	1231.1 (710.3)	373.3 (55.2)
MSC-WA*(6)	100	1223.2 (712.6)	373.3 (55.3)
WA*	100	1229.9 (713.4)	373.5 (55.7)

Figures in parentheses are standard deviations.

Table 2: Performance of MSC-WA* in puzzle.

Algorithm	Success Rate	Search Steps	Solution Length
MSC-WA* (1)	38	370374.0 (154466.0)	9495.0 (3554.9)
MSC-WA* (2)	97	190507.0 (142192.3)	9470.7 (2773.3)
MSC-WA* (3)	98	134848.0 (106015.8)	10180.7 (3433.3)
MSC-WA* (4)	100	120051.0 (93780.7)	11138.1 (3919.0)
MSC-WA* (5)	96	133266.0 (116221.0)	11584.8 (5316.8)
MSC-WA* (6)	99	157416.0 (146266.5)	11622.1 (6413.2)
WA*	10	189241.9 (118754.7)	3151.0 (619.7)

Figures in parentheses are standard deviations.

Table 3: Performance of MSC-RTA* in maze.

Algorithm	Success Rate	Search Steps	Solution Length
RTA*	100	7413.1 (10684.8)	363.5 (45.3)
MSC-RTA*(2)	100	7852.9 (11200.9)	359.1 (43.7)
MSC-RTA*(3)	100	7918.5 (11354.8)	358.5 (44.0)
MSC-RTA*(4)	100	8005.1 (11508.4)	358.3 (43.7)
MSC-RTA*(5)	100	8108.2 (11672.7)	358.1 (43.8)
MSC-RTA*(6)	100	8104.4 (11689.5)	358.2 (43.9)

Figures in parentheses are standard deviations.

Table 4: Performance of MSC-RTA* in puzzle.

Algorithm	Success Rate	Search Steps	Solution Length
RTA*	0		
MSC-RTA* (2)	100	155770.4 (126556.7)	44569.4 (29959.8)
MSC-RTA* (3)	100	93822.0 (60623.8)	23426.4 (14501.8)
MSC-RTA* (4)	100	120639.8 (97361.6)	21230.7 (15697.6)
MSC-RTA* (5)	100	181777.7 (182889.3)	24243.4 (31071.7)
MSC-RTA* (6)	100	210496.5 (214838.1)	24511.4 (27474.1)

Figures in parentheses are standard deviations.

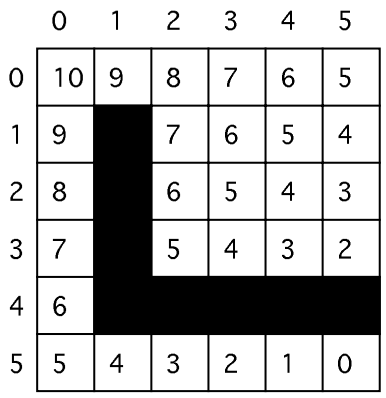

Figure 6: Maze and its heuristic values.

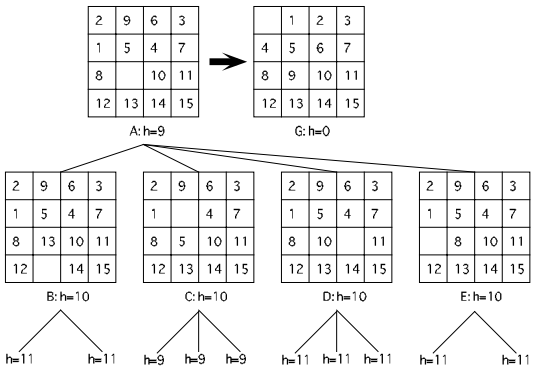

Figure 7: 15-puzzle and its partial state space graph.

in the following case. Let us assume that the size of $SG(s_1)$ is smaller than that of $SG(s_2)$. In this case, if MSC aborts the search in $SG(s_1)$ and restarts it in $SG(s_2)$, the performance will suffer. Moreover, if there are more alternatives than s_1 and s_2 and their h values look equally good, MSC may degrade its performance more. This is a reason for performance degradation when n is too large and for the MSC's superiority to ASC.

From the above discussion, MSC shows better performance if the heuristic function is misleading (Condition 1) and if MSC can abort the search (Condition 2). Now, let us analyze whether these conditions are satisfied with the maze and N-puzzle problems that are used in the experimental performance analysis.

Figure 6 shows a maze. Its size is 6×6, the initial state is (0,0), and the goal state is (5,5). The heuristic value of a state is the Manhattan distance to the goal state and it is written in the corresponding grid. These heuristic values satisfy Condition 1 from the following discussion. The initial state (0,0) has two child-states (0,1), which corresponds to s_2, and (1,0), which corresponds to s_1. Choosing (0,1) leads to the goal state and is apparently a correct choice. However, both of heuristic values are identical so this fact may cause a wrong choice. However, Condition 2 is not satisfied. This is because the h value of another candidate (0,1), which corresponds to s_2, is not less than all of the states in $SG(s_1)$ as shown in Figure 6. Hence, s_2 is not chosen from the commitment list regardless of n during the search in $SG(s_1)$ so MSC is not expected to be effective in such maze problems.

Figure 7 shows a partial state space graph of a 15-puzzle where the initial state is A and the goal state is G. This problem satisfies Condition 1. Namely, it is difficult to choose a right candidate because the heuristic values of the candidates B, C, D, and E, which are the child-states of the initial state A, are identical ($h = 10$). In this case, choosing B is a wrong choice as we discuss in the next subsection. In addition, this N-puzzle also satisfies Condition 2. That is because the h values of child-states B, which corresponds to s_1, are greater than that of C, which is corresponding to s_2. Hence, MSC eventually chooses s_2 from the commitment list, and returns to s_2 from $SG(s_1)$, so it is effective in N-puzzle problems. On the other hand, as shown in Figure 7, since N-puzzle has a number of states with an identical h value, MSC, with large n, or ASC expands the search front too widely, and degrades the performance or consume all the memory before reaching a goal state.

6.2 Serializable Subgoals

MSC shows a dramatic performance improvement at N-puzzles. As shown in Figure 5, the performance difference between SSC and MSC depends on the size of $SG(s_1)$ because SSC takes a long time for finishing the search in $SG(s_1)$ but MSC can abort the search at an early stage.

We here explain that the size of $SG(s_1)$ in N-puzzles is large by introducing the concept of serializable subgoals [7],[11]. A problem has serializable subgoals iff the goal can be divided into a set of subgoals, and there exists an ordering among the subgoals such that the subgoals can always be solved sequentially without ever violating a previously achieved subgoal in the order [7]. For example, if we sort the bottom row of the 15-puzzle as a subgoal, then we can always solve the rest of the problem without disturbing the bottom row. Furthermore, if we sort the right column as the next subgoal, the original 15-puzzle problem is reduced to an 8-puzzle. As in Figure 8, we can assign

a level to each of subgoals depending on the order of achievement.

Hence, in the N-puzzle, it is better to choose an expansion state so that it does not destroy the subgoal that have already been achieved. However, the heuristic value, which is the sum of the Manhattan distances of misplaced tiles, does not reflect this property. For example, in Figure 7, since the initial state has already achieved the level-2 subgoal, choosing state B will destroy it. However, since the heuristic values of four child-states are identical, it is difficult to avoid the destruction. Once the algorithm destroys the subgoal, it takes many steps to rebuild it, so in choosing state B the performance is degraded. However, MSC can return to another candidate from the commitment list so the wrong choice of B is not so fatal.

To support our discussion, we present how subgoals are destroyed in 24-puzzles in Table 5. In MSC search algorithms, we define the level of achieved subgoal as the highest one in the commitment list. Precisely, the subgoal level at time T is defined as $SL^T = max_{s \in \mathrm{COMMITMENT}^T} SL(s)$ where $SL(s)$ is the subgoal level of state s. Please note this is not always the same as the subgoal level of expanded state at time T. Hence, the average steps to decrease subgoal level is calculated from the whole search steps divided by the number of events that decrease the subgoal level.

This table shows that more states with higher level subgoal are preserved in the commitment list as n increases. Therefore, even if the algorithm chooses a state which destroys a subgoal temporally, it will easily choose another state which does not destroy the subgoal. This effect is strengthened by enlarging n.

Table 5: Subgoal destruction in 24-puzzle.

Algorithm	Average steps to decrease subgoal level
RTA*	97.9
MSC-RTA*(2)	223.0
MSC-RTA*(3)	665.3
MSC-RTA*(4)	1327.3
MSC-RTA*(5)	1972.4
MSC-RTA*(6)	2719.3

7 Related Works

One method to decrease the risk of making wrong choices is a method called *cooperative search*, in which multiple problem solvers (agents) concurrently solve an identical problem. Since there are multiple agents, if at least one agent can make correct choices, a solution can be obtained in a reasonable amount of time. Cooperative search methods have been applied to constraint satisfaction problems [2, 4] and state-space search problems [6, 5].

One drawback to these cooperative search methods is that if agents have to make critical choices repeatedly, those agents that made wrong choices early are unable to contribute to the ongoing search process. [14] presents the multiagent real-time A* algorithm (MRTA*) with a GA-like selection mechanism. In this algorithm, a state-space search problem is solved concurrently by multiple agents. Each agent executes the RTA* algorithm, and periodically reproduces offspring stochastically based on its fitness defined by the heuristic value of its current state. Experimental evaluation results show that this algorithm is very effective in N-puzzles. Our work was inspired by [14], and we show that much simpler algorithms can obtain similar results. While agents tend to do redundant actions, i.e., expanding identical states in MRTA*, our algorithms in this paper avoid the redundance. For example, in [14], it is reported that the MRTA* algorithm with the selection mechanism requires 35452.7 steps when five agents[3] solve the 48-puzzle problems concurrently (177263.5 steps in all). On the other hand, MSC-RTA* (where the length of the commitment list is 3) requires only 93822.0 steps.

The beam search algorithm [1] is a classical method for focusing/limiting the search effort. In a beam search, expanded nodes that are not included in the scope of the search attention are completely pruned. As a result, the algorithm completeness cannot be guaranteed so, for example, it can not solve many of maze problems. In this paper, on the other hand, the completeness of the original algorithms (RTA* and WA*) is preserved.

A more modern method for focusing/limiting search efforts is a family of Limited Discrepancy Search algorithms [3, 10]. These algorithms are based on a depth-first search algorithm, and try to incrementally broaden the search range by using heuristic information. These algorithms are suited to the problems with a known upper-bound of the search depth, such as constraint satisfaction problems.

One way for reducing the required memory size in the A* algorithm is the SMA* algorithm [13], which removes unpromising nodes from the OPEN list. This memory-reducing method and the idea of commitments introduced in this paper are not mutually exclusive, and can be used simultaneously.

[3] This setting gives the best result both in the total steps and the steps for each agent.

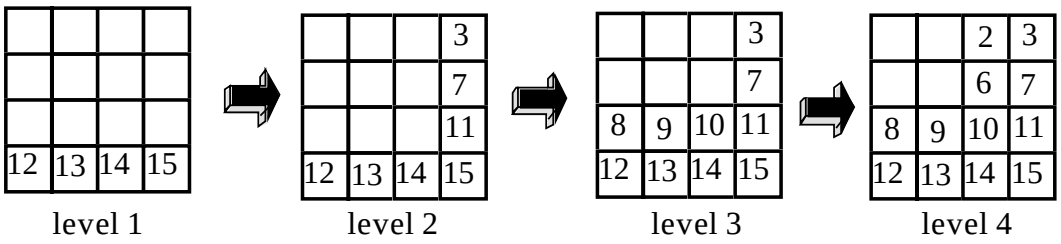

Figure 8: Achievement of subgoals.

The recursive best-first search (RBFS) algorithm [9] can run in linear space. However, as reported in [9], this algorithm does not work well when $f(s) = h(s)$. This is because the RBFS tends to explore all possible paths to a given node, and the number of duplicate nodes explodes as the search depth increases.

8 Conclusion

In this paper, we introduce the Multi-State Commitment method, which limits the search range, to two semi-optimal solution search algorithms, WA* and RTA*. We then evaluate their performance in maze and N-puzzle problems and show a dramatic performance improvement in N-puzzle problems. We also present required conditions for performance improvement that are satisfied in N-puzzles and the relation to serializable subgoals.

The MSC method which uses a commitment list is a generic method to improve search efficiency, so in the future, we will apply this method to other kinds of algorithms and problems. In this paper, we do not discuss the optimal value of the parameter n. We guess this depends on the character of problem and leave it as an open problem. It is also interesting to provide a dynamic method to adjust n during a search process.

References

[1] Bisiani, R. 1992. Beam search. In Shapiro, S. C., ed., *Encyclopedia of Artificial Intelligence*. New York: Wiley-Interscience Publication. 1467–1468.

[2] Clearwater, S. H.; Huberman, B. A.; and Hogg, T. 1991. Cooperative solution of constraint satisfaction problems. *Science* 254:1181–1183.

[3] Harvey, W. D., and Ginsberg, M. L. 1995. Limited discrepancy search. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, 607–613.

[4] Hogg, T., and Williams, C. P. 1993. Solving the really hard problems with cooperative search. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, 231–236.

[5] Kitamura, Y.; Teranishi, K.; and Tatsumi, S. 1996. Organizational strategies for multiagent real-time search. In *Proceedings of the Second International Conference on Multi-Agent Systems*, 150–156.

[6] Knight, K. 1993. Are many reactive agents better than a few deliberative ones? In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, 432–437.

[7] Korf, R. E. 1988. Search in AI: A Survey of Recent Results. In Shrobe, H. E., ed., *Exploring Artificial Intelligence* . Morgan-Kaufmann.

[8] Korf, R. E. 1990. Real-time heuristic search. *Artificial Intelligence*, 42(2-3):189-211.

[9] Korf, R. E. 1993. Liner-space best-first search. *Artificial Intelligence*, 62(1):41-78.

[10] Korf, R. E. 1996. Improved limited discrepancy search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, 286–291.

[11] Newell, A. and Simon, H. A. 1972. *Human Problem Solving*. Prentice-Hall.

[12] Pearl, S. 1984. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley Publishing Company.

[13] Russel, S. 1992. Efficient memory-bounded search method. In *Proceedings of the Tenth European Conference on Artificial Intelligence*, 1–5.

[14] Yokoo, M., and Kitamura, Y. 1996. Multiagent real-time-a* with selection: Introducing competition in cooperative search. In *Proceedings of the Second International Conference on Multi-Agent Systems*, 409–416.