Real-time Adaptive GPU multi-agent path planning

Ugo Erra* Giuseppe Caggianese*

27 July, 2011

DRAFT 27 July, 2011 - 11:32

1 Introduction

Many types of computer games involve player and non player characters moving over terrain. In some types of game, the player directly controls a main character while the non player characters are controlled by path planning algorithms. In real time strategy games the player doesn't control any characters directly. Instead, the player selects a group of characters (or agents) and then selects a target position with the mouse. The target position can be in a known position (a position that the agents have observed before) or an unknown position. The agents then have to find their own way to the target position. The characters do not know the whole terrain in advance, instead they observe a certain range around them and remember the positions they've observed for future use. If they observe that their current trajectory is blocked after they have started moving, they have to search for another path. A fast path planning algorithm is therefore essential for the agents to move smoothly around obstacles.

A*[1] is the most famous algorithm for finding cost-minimal paths in state spaces, which are usually represented as graphs. Given a start state (start node) and a goal state (goal node) A* finds the least-cost path by using a distance-cost heuristic function to determine the order in which the search visits states. The search performed by A* is ideal for off-line artificial intelligence applications, but it is not suitable for computer games where agents have to search paths in real-time.

This work proposes an efficient multi-agent planning approach for the GPU.

*Dipartimento di Matematica e Informatica, Università della Basilicata, Potenza, Italy

The implementation is based on a previous algorithm called Real-Time Adaptive A* (RTAA*)[2]. In RTAA*, the search is restricted to a small part of the state space that can be reached from the current state using a single A* search episode. For each search episode, the agent determines a local search space, searches it, updates the distance-cost heuristics, and moves along the resulting trajectory. The agent repeats this process until it reaches a goal state. RTAA* is efficient in real-time applications but has a major drawback, the large amount of memory required for each agent, limits the number of simultaneous searches. Our approach reduces the amount of memory for each agent, enabling the design of a parallel implementation of RTAA* for the GPU architecture. This offers a simple and powerful way to accelerate simultaneous path planning in real-time applications such as computer games and robotics.

2 Core Method

Before explaining the GPU implementation, we give a brief overview of the A* and RTAA* algorithms. In A* (Algorithm 1), for every state s, the user supplies a heuristic $h[s]$ that estimates the goal distance, which is the cost of a minimal path from the state s to a goal state. Classical heuristics are based on Manhattan, diagonal, or Euclidian distance calculations. During its execution, A* maintains two values, $g[s]$ and $f[s]$. The value $g[s]$ is the smallest cost of any discovered path from the start state s_{start} to state s. The value $f[s] = g[s] + h[s]$ estimates the distance from s_{start} to the goal state via state s. The algorithm maintains two lists, the open list and the closed list. The open list is a priority queue and contains the most recently discovered states. Initially it contains only the start state s_{start}. The closed list contains the expanded states, those from which all adjacent states have been explored and inserted into open list. At each iteration, A* removes the state s with the smallest $f[s]$ value from the open list. If state s is a goal state, it terminates. Otherwise, it explores the adjacent states and updates the g-value of each visited state. If the g-value decreases, it updates the g-value and the corresponding f-value in the open list. It then repeats the process. Finally, the g-value of every visited state s will be the distance from the start state s_{start} to state s.

For real-time applications, A* has two main drawbacks. The first is the computational time required to perform a search from the start state to the goal state. The second disadvantage relates to memory footprint during the

DRAFT 27 July 2011-11:32

execution of the algorithm. Each agent must store and update the $h[s]$, $g[s]$, and $f[s]$ values for each state s. This makes A* unsuitable for multi-agent path planning in large state spaces when memory is limited.

Algorithm 1 A*

```
 1: ClosedList ⇐ ∅
 2: OpenList ⇐ start
 3: g[start] ⇐ 0
 4: h[start] ⇐ HeuristicEstimate(start, goal)
 5: f[start] ⇐ h[start]
 6: while OpenList ≠ ∅ do
 7:    x ⇐ ExtractLowerValue(OpenList)
 8:    if x = goal then
 9:       return  ReconstructPathFrom(goal)
10:    end if
11:    for all s ∈ NeighborNodes(x) do
12:       newg ⇐ g[x] + Cost(x, s)
13:       if s ∈ OpenList and newg < g[s] then
14:          Remove(OpenList, s)
15:       end if
16:       if s ∈ ClosedList and newg < g[s] then
17:          Remove(ClosedList, s)
18:       end if
19:       if s ∉ OpenList or s ∉ ClosedList then
20:          parent[s] ⇐ x
21:          g[s] ⇐ newg
22:          h[s] ⇐ HeuristicEstimate(s, goal)
23:          f[s] ⇐ g[s] + h[s]
24:          Add(OpenList, s)
25:       end if
26:       Add(ClosedList, x)
27:    end for
28: end while
```

Real-Time Adaptive A* (Algorithm 2) is a real-time heuristic search method that chooses its local search spaces in a very fine-grained way. The main idea is to update the heuristics of all states in the local search space very quickly and to save the heuristics to speed up future A* searches. This approach uses a variable called `lookahead`, which specifies the largest number of states to expand during an A* search. After the A* search, we define $\overline{s}$ to be the state that was about to be expanded when the A* search terminated. At this point, RTAA* updates the heuristic of all the expanded states s in the closed list by setting $h[s] = g[\overline{s}] + h[\overline{s}] - g[s]$. RTAA* then executes the plan along the trajectory found by the A* search until state $\overline{s}$ is reached. Koenig et al. [2] have proven that this heuristic becomes more informed over time and that it is consistent, ensuring a trajectory of smaller cost for a given time-limited search episode.

DRAFT 27 July, 2011- 11:32

Algorithm 2 RTAA*

1: *lookahead* $\Leftarrow$ any integer greater than zero
2: *movements* $\Leftarrow$ any integer greater than zero
3: **while** $s_{curr} \neq goal$ **do**
4: A*() {Expand lookahead states in A*}
5: **if** $\overline{s} = FAILURE$ **then**
6: **return** FAILURE
7: **end if**
8: **for all** $s \in$ ClosedList **do**
9: $h[s] \Leftarrow g[\overline{s}] + h[\overline{s}] - g[s]$
10: **while** $s_{curr} \neq \overline{s}$ **and** $movements > 0$ **do**
11: $a \Leftarrow$ ActionOnTrajectory()
12: $s_{curr} \Leftarrow succ(s_{curr}, a)$
13: $movements \Leftarrow movements - 1$
14: ChangeCosts()
15: **end while**
16: **end for**
17: **end while**

The key aim in designing and implementing a RTAA* multi-agent path plan in the GPU is to reduce the memory required for $g[s]$, $h[s]$, and $g[s]$ for all states s. Table 1 shows the main input and output variables handled by RTAA* for each agent. Note that, for each search episode, we need a Start state variable, and the queues for Open List, Closed List, and Parent List that must be sufficiently large to handle the number of lookahead states. Moreover, for each search episode we need a Goal state variable and the arrays G_{cost}, H_{cost}, and F_{cost} to keep updated the values of any discovered path $g[s]$, estimated cost distance $h[s]$, and the cost of the estimated path $f[s]$, respectively, of each state s. The amount of memory required for G_{cost}, H_{cost}, and F_{cost} depends on the number of states.

The proposed implementation is based on the observation that after each search episode G_{cost}, H_{cost}, and F_{cost} values only in the surrounding area of an agent's current position are updated. In addition, as the agent moves along the path toward the goal state, values related to explored states will not be required in the current search episode. Thus, we do not maintain these values after a certain number of search episodes. In our GPU implementation, we exploit the variable `lookahead` in order to take into account only those values in the surrounding area of the agent's current position and then to reduce the memory footprint required for each agent.

DRAFT 27 July, 2011 - 11:32

INPUT			
Used	**Variable**	**Description**	**Init**
PER SEARCH EPISODE	START	Start state	USER
	OPEN LIST	List of discovered states	ZERO
	CLOSED LIST	List of expanded states	ZERO
	PARENT LIST	List of successors	ZERO
FOR ALL SEARCH EPISODES	GOAL	Goal state	USER
	G_{cost}	Cost of any discovered path $g[s]$, for each state s	∞
	H_{cost}	Estimated goal distance $h[s]$, for each state s	Heuristic function
	F_{cost}	Cost of estimated path $f[s]$, for each state s	ZERO
OUTPUT			
Used	**Variable**	**Description**	**Init**
PER SEARCH EPISODE	$\bar{s}$	Start state for the next A* search	ZERO
	PATH LIST	Path state list	ZERO

Table 1: INPUT and OUTPUT data for RTAA*. The variables in a search episode keep state expansion information for the current A* search.

3 Implementation

In this section, we describe a path planning system for many thousands of agents that uses RTAA*. NVIDIA's CUDA is the platform chosen for exploiting data parallelism in the GPU. The following subsections discuss parallel pathfinding implementation on the GPU.

3.1 Grid map

Our implementation is suitable for games that are based on a grid map. In these games, the world is subdivided into small regular zones called tiles. Each tile represents a state s of the state space and is connected to all nearby tiles. These connections form a tile-graph as illustrated in Figure 1.

The cost of moving from a tile to each of its neighbours is specified by an integer. This can be used to model terrain elements that are difficult or impossible to pass, for example hills and lakes. A common metric used on grid maps, which we adopt for this work, is the Manhattan distance. The memory footprint for each grid map is $4 \times h \times w$ words, where h and w are the height and width of the map, respectively. At the initial phase of planning, the grid map is copied from host memory to the device's global memory region and processed by a grid of threads.

DRAFT 27 July, 2011 - 11:32

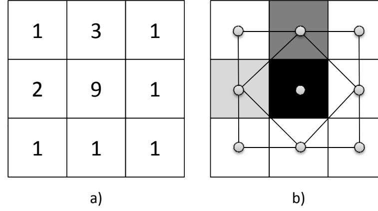


Figure 1: a) Grid map. An integer indicates a terrain element. b) Tile-graph. A light grey tile indicates a hill, the dark grey a mountain, and the black tile an obstacle which is unreachable.

3.2 Lookahead and movements array

In order to perform searches for many thousands of agents in parallel we need a way to reduce the working set per thread. Our solution exploits the `lookahead` parameter of the RTAA* algorithm, and this parameter allows us to store for each agent the G_{cost}, H_{cost}, and F_{cost} values only for a limited area surrounding the current agent position. This area is tracked by using two overlay arrays, called the **lookahead array** and the **movements array** (Figure 2).

The lookahead array is centred on the agent start position at the beginning of a search episode and tracks all the tiles that are discoverable during the search episode. Its size is $(\texttt{lookahead} \times 2 + 1)^2$ because during the search episode an agent can explore, at the most, `lookahead` tiles in all directions. These tiles are inserted in the queues Open List and Closed List as illustrated in Section 2. Thus, the memory required for Open List and Closed List is $(\texttt{lookahead} \times 2 + 1)^2$ words. However, the memory required for Parent List is `lookahead` words, because for each search episode we need only to store lookahead expanded tiles from which all adjacent tiles have been explored and inserted in the Open List.

The movements array is used to keep track of those tiles that are explorable during a certain number of successive search episodes. Each agent, by using the movement arrays, keeps a cost list for G_{cost}, H_{cost}, and F_{cost} only for those tiles contained in the movements array. When an agent performs a search episode, it stores and updates $g[s]$, $h[s]$, and $f[s]$, where s is a tile in the movements array. This array enables us to avoid maintaining an array of the same size as the grid map to store G_{cost}, H_{cost}, and F_{cost} values for each agent. The inspiration for this arises from the observation that an agent rarely requires or updates the values $g[s]$, $h[s]$, and $f[s]$ for some tile

DRAFT 27 July, 2011 - 11:32

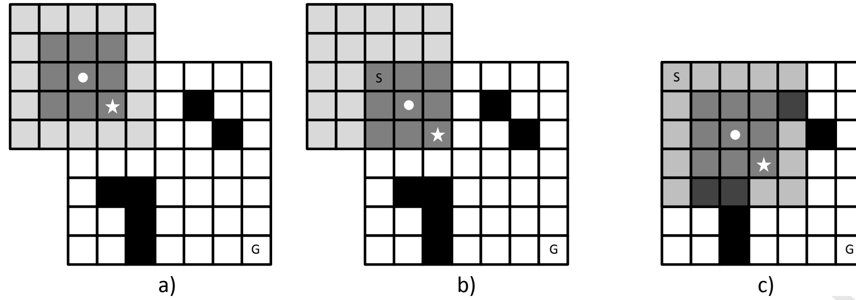


Figure 2: The dark-grey array is the lookahead array. The light-grey array is the movements array. Behaviour of the arrays during a search episode with `lookahead=1`. The white dot is the start tile. The white star is the next start tile. a) The lookahead array position is always centred on the start position. b) After each search episode, the lookahead array will move according to the next start tile. As long as the movements array contains the lookahead array, it does not change its current position. c) When the lookahead array moves out from the movements array then we must shift both arrays and centre them on the new start tile.

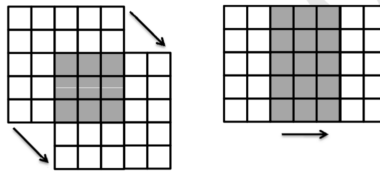


Figure 3: Two examples of movements array motion. The grey cells show that values $g-$, $h-$, and $f-$ are still valid when the movements array is moved. These values must be retained and then copied.

s discovered in past search episodes. In our implementation the size of the movement array is $(\texttt{lookahead} \times 4 + 1)^2$. Thus, the memory layout required for the three cost lists is $(\texttt{lookahead} \times 4 + 1)^2 \times 3$ words.

According to the positions of the lookahead and movements arrays for each agent, we update only those values in the agent's surrounding area. Initially, both arrays are centred on a start tile (Figure 2a). After a search episode the lookahead array will move according to the next start tile (Figure 2b). As long as the movements array contains the lookahead array, it does not change its current position. When the lookahead array moves out from the movements array, both arrays must be moved and centred on a new start tile (Figure 2c).

Note that before the movements array moves into a new position, the same values may still be valid and must be retained for the next search episode.

In Figure 3, we show valid values in the movements array between two positions. Based on the direction these values must be copied to another part of the movements array. This operation is performed by simple read and write operations inside the same block of memory with negligible overhead.

3.3 The working set

The implementation of a search episode is designed using four kernels that have several inputs and two outputs. The input for all agents is a grid map. Whereas, the inputs for each agent include:

- A Start and a Goal tile. Initialized by the user.
- A NextStartTile value to keep track of the next starting tile $g[\bar{s}]$.
- A State value to maintain the state of the search.
- The Open List, Parent List, and Closed List. Initialized to zero.
- A cost list of updated heuristics H_{cost}. Initialized the first time by the host.
- A cost list of discovered paths G_{cost}. Initialized to ∞.
- A cost list of estimated paths F_{cost}. Initialized to zero.
- A Counter value to keep track of the current search episode.
- A list of values $search(s) = i$ if state s has been generated last by the ith A* search. This list is used to check if a tile has been generated in the previous search episodes. In the case $search(s) \neq$ counter we set $g[s] = \infty$, enable us to rediscover this tile and improve the trajectory. Initialized to zero.

The pair of outputs for each agent are:

- A trajectory found in the last search episode without obstacles.
- The start tile $\bar{s}$ for the next A* search.

All input data structures reside in global memory. Static data structures, e.g., the grid map, are kept in cached constant global memory. While any modifiable data structures are kept in non cached read-write global memory locations. All data structures are stored in an efficient collection of

DRAFT 27 July, 2011 - 11:32

	Map	States	Memory KB
T0	20×20	400	1600
T1	40×40	1600	6400
T2	80×80	6400	25600
T3	160×160	25600	102400
T4	320×320	102400	409600
T5	1024×1024	1048576	4194304

	Agents	Blocks	Memory MB
G0	128	1	0.39
G1	8192	64	24.72
G2	28800	225	86.69
G3	80000	625	241.39
G4	139392	1089	420.61
G5	294912	2304	889.88

Table 2: Left: the size of the grid maps and the GPU memory footprint for each one. Right: the number of agents (threads), the number of thread blocks (128 threads per block), and the memory footprint for each group of agents with `lookahead=3`. In the worst-case scenario (T5 grid map and 294912 agents) the total amount of memory used is below 5MB.

Structure-of-Arrays (SoA) that improves the probability of coalesced memory transactions across a half-warp.

3.4 The CUDA kernels

Each agent is processed by a thread on the GPU. In order to improve the distribution of resources in the streaming multiprocessors, we implement the search episode process in four sequential CUDA kernels:

- `InitializeArray`: moves the lookahead array in the current tile $\overline{s}$ and moves the movements array when necessary.
- `InitializeSearch`: initializes $g[\overline{s}]$ and $f[\overline{s}]$ for the current tile $\overline{s}$ and inserts it in the Open List.
- `SearchEpisode`: performs the A* search from $\overline{s}$ expanding lookahead tiles. The priority queue is implemented in a similar way to that proposed in [3].
- `UpdateAndCheck`: updates the heuristics $h[s]$ of the tiles s contained in the Closed List, creates the path using the Parent List, and checks the presence of obstacles along the path.

4 Results

In this section, we show the results of two types of experiments. The first experiment demonstrates the efficiency of our approach, the second is related

DRAFT 27 July, 2011 – 11:32

	Average Path Lengths	Average Search Episodes
T0	10	7
T1	20	13
T2	38	27
T3	77	53
T4	151	105
T5	478	336

Table 3: Average path lengths and average search episodes in planning all groups of agents on each grid map.

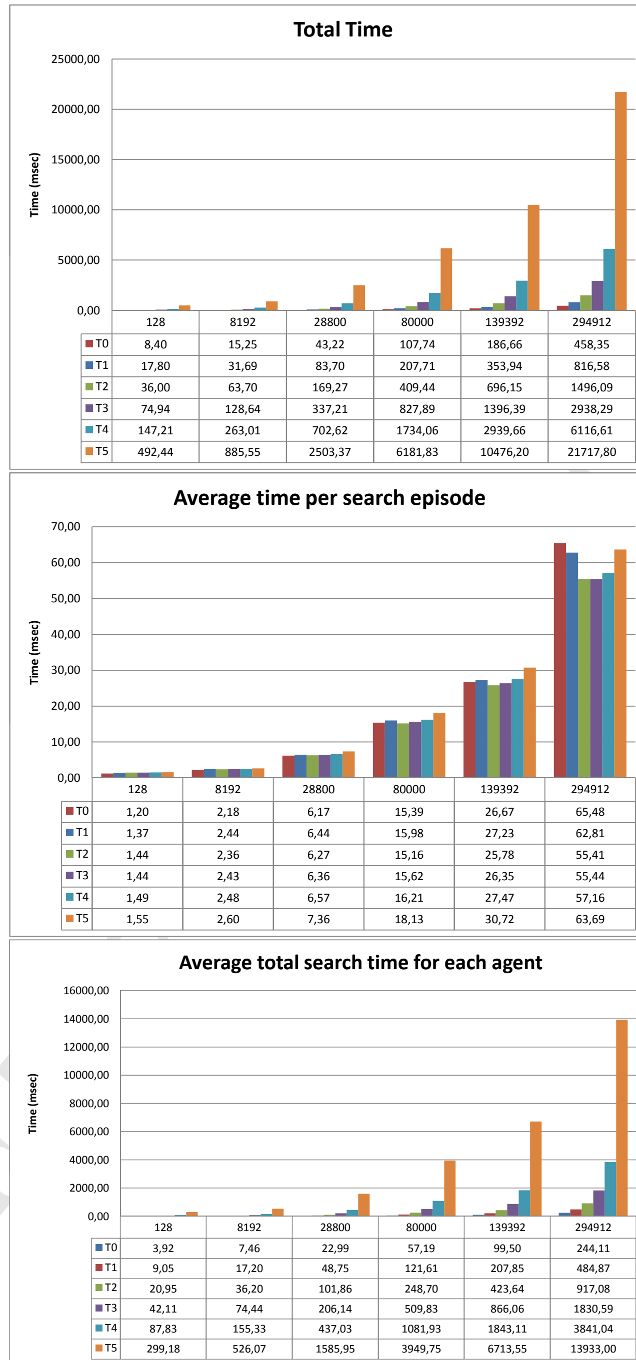
to the quality of the trajectory found using the lookahead and movements arrays.

The experiments were performed on six grid maps of sizes ranging from 20×20 to 1024×1024. Start and goal tiles were randomly chosen, and `lookahead` was set to 3. For each grid map, we launched several groups of agents, of size 128 to 294912, and each CUDA block had 128 threads. The GPU memory footprint for the grid maps and agent groups can be seen in Table 2. Our tests were performed on an Intel Core i7 CPU 1.6GHz, NVIDIA Fermi GTX 470 1.28GB, Windows 7. All the kernels were written in CUDA 2.1 and Microsoft's Visual C++ 2008 compiler.

Table 3 lists the average path lengths and the average number of search episodes that occurred for all groups of agents. Figure 4 shows the performance of the GPU. The absolute running time for the benchmarks executed on the GPU ranges from 458 milliseconds for T0 to 21717 milliseconds for T5 and an average time per search episode of 65 milliseconds for T0, up to 63 milliseconds for T5. The smallest value, as observed for G5, arose because as the number of search episodes in T5 increases, the average time per search episode decreases, although the number of agents is greater than G0 (see Table 3). At the bottom of Figure 4, we can see the total average time taken for agent to reach its goal. Figure 5 supports the GPU CUDA performance scale compared to multithreading CPU implementation in running one-, two-, and four-threads.

A GPU implementation of the A* algorithm has been presented by Bleiweiss [3]. Although the results of this work cannot be directly compared owing to the different hardware generation used, we can note the difference in terms of memory footprint. In Bleiweiss's work, with a map of 340 nodes and 115600 agents, the working set memory (about 1.5GB) exceeds the available GPU global memory, and searches are thereby broken into multiple pathfinding passes, each one responsible for a subset of the total agents. In our work,

DRAFT 27 July, 2011 - 11:32



	128	8192	28800	80000	139392	294912
T0	8,40	15,25	43,22	107,74	186,66	458,35
T1	17,80	31,69	83,70	207,71	353,94	816,58
T2	36,00	63,70	169,27	409,44	696,15	1496,09
T3	74,94	128,64	337,21	827,89	1396,39	2938,29
T4	147,21	263,01	702,62	1734,06	2939,66	6116,61
T5	492,44	885,55	2503,37	6181,83	10476,20	21717,80

	128	8192	28800	80000	139392	294912
T0	1,20	2,18	6,17	15,39	26,67	65,48
T1	1,37	2,44	6,44	15,98	27,23	62,81
T2	1,44	2,36	6,27	15,16	25,78	55,41
T3	1,44	2,43	6,36	15,62	26,35	55,44
T4	1,49	2,48	6,57	16,21	27,47	57,16
T5	1,55	2,60	7,36	18,13	30,72	63,69

	128	8192	28800	80000	139392	294912
T0	3,92	7,46	22,99	57,19	99,50	244,11
T1	9,05	17,20	48,75	121,61	207,85	484,87
T2	20,95	36,20	101,86	248,70	423,64	917,08
T3	42,11	74,44	206,14	509,83	866,06	1830,59
T4	87,83	155,33	437,03	1081,93	1843,11	3841,04
T5	299,18	526,07	1585,95	3949,75	6713,55	13933,00

Figure 4: GPU performance for all group of agents run over all six grid maps. We measured the total time to search the paths for all six groups of agents, the average time per search episode, which is critical for real-time applications, and average total search time per agent, which measures parallelism efficiency on the GPU.

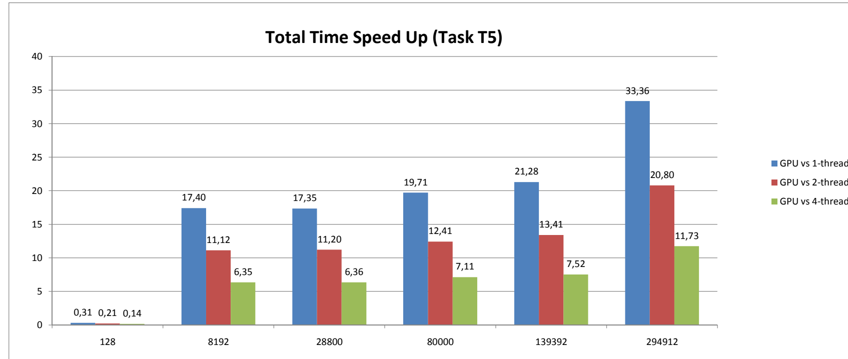


Figure 5: T5 performance of the NVIDIA GTX 470 GPU compared to three multi-thread CPU versions, using an Intel Core i7 CPU 1.6 GHz.

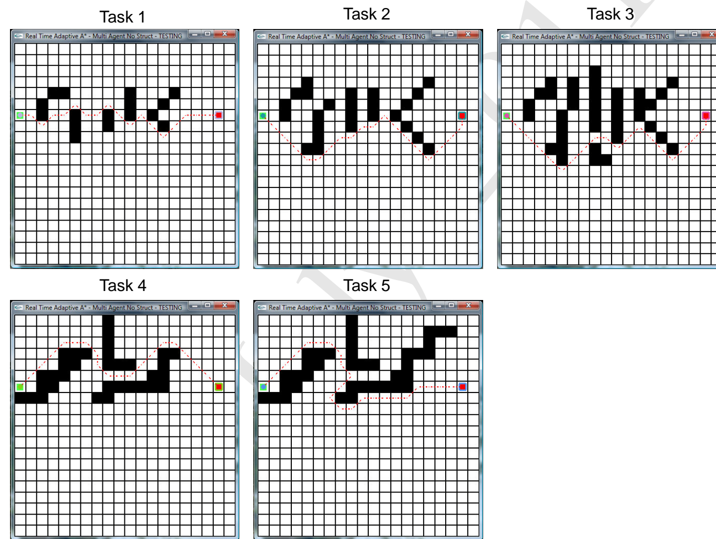


Figure 6: In each task the start position is on the left and the end position is on the right. The sketched line is the optimal path.

a comparable configuration with T0 grid map and G4 agent configuration, the memory footprint is below 500MB.

The second type of experiment concerns the length of the path obtained from our approach owing to the introduction of the lookahead and movements arrays. For this purpose, we use the five tasks shown in Figure 6. For each task, we perform A* and RTAA* with `lookahead` set to 3–7. Table 4 shows the results of this experiment. Note that, for task T1, the length of the path

is the same, independent of the value of `lookahead`. The worst case is task T3 where `lookahead` equals 7. In this scenario, the path length is 19 for A* and 34 for RTAA*. The results of this experiment suggest that in some cases the approach finds paths whose lengths are worse than the optimal solution though the difference from the optimal path length is not significant. This deficiency is compensated for in terms of efficiency as shown in performance experiments. Tuning the `lookahead` parameter allows the user to trade off speed against path optimality. For example, in a real-time application, speed is the highest priority and suboptimal paths may be acceptable.

Pathfinding		Task 1	Task 2	Task 3	Task 4	Task 5
A*	Length	**18**	**19**	**19**	**19**	**23**
RTAA*-3	Length	18	28	33	26	31
	Search episodes	10	18	18	13	16
RTAA*-4	Length	18	23	40	20	29
	Search episodes	6	10	23	7	12
RTAA*-5	Length	18	25	23	20	27
	Search episodes	5	10	8	6	9
RTAA*-6	Length	18	21	32	20	26
	Search episodes	5	8	14	5	6
RTAA*-7	Length	18	19	34	22	27
	Search episodes	4	4	11	6	7

Table 4: For all the tasks, the A* row reports the least-cost paths, while GPU RTAA* rows report the path lengths and the search-episode counts with increasing `lookahead` values.

4.1 Results and Conclusions

In this chapter, we have shown that an implementation based on the Real-Time Adaptive A* algorithm fits well with the GPU parallel architecture. By using a limited memory footprint per thread, it offers a simple and powerful way to plan trajectories for many thousands of agents in parallel. The implementation manages only static and known obstacles reported in the grid map. In future research, we plan to implement the management of unknown obstacles that occur in the grid map. Once an agent recognizes the presence of an unknown obstacle, it will report it in the grid map and share this information with other agents. The unknown obstacle will become visible and will be taken into account in subsequent search episodes. This approach can easily be extended to cope with the presence of dynamic obstacles.

DRAFT 27 July, 2011 11:32

References

[1] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *SIGART Bull.*, 4(37):28–29, 1972.

[2] Sven Koenig and Maxim Likhachev. Real-time Adaptive A*. In Peter Stone and Gerhard Weiss, editors, *AAMAS '06: Proceedings of the fifth International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 281–288, New York, NY, USA, 2006. ACM.

[3] Avi Bleiweiss. GPU Accelerated Pathfinding. In Dieter Fellner and Stephen Spencer, editors, *GH '08: Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics hardware*, pages 65–74, Aire-la-Ville, Switzerland, 2008. Eurographics Association.

DRAFT 27 July, 2011, 11:32