Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling

Moving Target Search with Subgoal Graphs*

Doron Nussbaum and **Alper Yörükçü**
School of Computer Science
Carleton University

Abstract

Moving Target Search (MTS) is a dynamic path planning problem, where an agent is trying to reach a moving entity with a minimum path cost. Problems of this nature can be found in video games and dynamic robotics, which require fast processing time (real time). In this work, we introduce a new algorithm for this problem - the Moving Target Search with Subgoal Graphs (MTSub). MTSub is based on environment abstraction and uses Subgoal Graphs to speed up searches without giving up cost minimal paths. The algorithm is optimal with respect to the knowledge that the agent has during the search. Experimental results show that MTSub meets the requirement of real time performance (e.g., 5 microseconds per step). Compared to G-FRA*, which is the best known dynamic algorithm so far, MTSub is up to 29 times faster in average time per step, and 186 times faster in maximum time per step. MTSub also compares fairly well against MtsCopa. Although in this case MTSub is up to 3.89 times slower in average response time and up to 6.81 times slower in maximum response time, it performed much better than MtsCopa in the processing phase - up to 220,000 times faster and requiring up to 44 times less space.

Introduction

A large number of applications, in video games, robotics and virtual simulations, require agents to plan their path, not only to a stationary target but also with respect to a moving target. When a change in a target location is observed, agents must override their plan during execution and react to the change in the target location. Namely, the current path has to be updated in order to obtain a better path. The speed with which these operations are executed is important because agents must decide where to move next in real time. The quality of the updated solutions is also crucial, because it directly affects the cost of reaching the target.

Moving Target Search (MTS) is a path planning problem where an agent attempts to reach a moving target (Ishida and Korf 1991). In this paper, we focus on moving target search where at all times, the agent has full knowledge of the search environment, which is static, and the target position. This variant of MTS often arises in robotic applications that run in a known environment and in video games.

There are generally two kinds of strategies for solving MTS problems with search algorithms (Sun, Yeoh, and Koenig 2010):

- *Offline techniques* (Moldenhauer and Sturtevant 2009; Hahn and MacGillivray 2006; Vieira, Govindan, and Sukhatme 2008; 2009) consider all possible locations of the agent and the target in the environment to determine the best plan, prior to movement of the agent. These techniques are not applicable to large environments due to large numbers of possible future actions of a target.

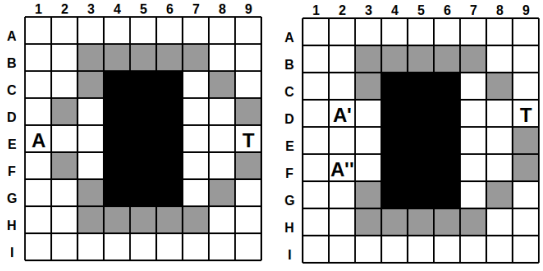
- *Online techniques* find a solution according to current information and update the existing plan when changes occur. They solve a series of path planning problems during plan execution to react to the target movement. Earlier online algorithms (Ishida and Korf 1991; 1995) find a prefix of the path with a bounded search. Therefore, they can work under strict time bounds. However, the cost of reaching the target location is high since they make shortsighted decisions and it is hard to determine the existence of a path between an agent and a target. Subsequent online algorithms (Sun, Yeoh, and Koenig 2009; 2010; Sun et al. 2012) find a complete path from the agent towards the target and update the path during the search. Fringe Retrieving A* (FRA*) (Sun, Yeoh, and Koenig 2009) and Generalized Fringe Retrieving A* (G-FRA*) (Sun, Yeoh, and Koenig 2010) are two algorithms that calculate cost minimal, complete paths between the current positions of the agent and the target. The overall cost of reaching the target using G-FRA* or FRA* is smaller than that of earlier online algorithms that find only a prefix of the path. Nevertheless, they cannot run under strict time bounds. Incremental Any time Repairing A* (I-ARA*) (Sun et al. 2012) can compute paths between current locations of the agent and the target under user defined time limits. However, the paths may not be cost minimal. Thus, the cost of reaching the target is higher (Botea et al. 2013) than algorithms that find cost minimal paths between current positions of the agent and the target. A more recent paper introduced Moving Target Search with Compressed Databases (MtsCopa) (Botea et al. 2013) which exploits

*The research is partially supported by the Natural Sciences and Engineering Research Council of Canada.

preprocessing to speed up online search without sacrificing cost minimal paths between current locations of the agent and the target. Unfortunately, preprocessing time and space requirements of the algorithm are substantially high (e.g., over eleven hours of preprocessing time and 23MB of space are required for a 320 x 320 game map). The performance of MtsCopa against latest algorithms for fast first move queries is discussed in (Strasser, Harabor, and Botea 2014).

We introduce an innovative, incremental search algorithm that uses environment abstraction to shorten response time of the agent without giving up cost minimal paths, Moving Target Search with Subgoal Graphs (MTSub). MTSub uses Subgoal Graphs (Uras, Koenig, and Hernández 2013) to create an abstract search environment and determine the path by solving numerous instances of the path planning problem incrementally. Subgoal Graphs restricts the applicability of MTSub to eight connected grids but this also lets MTSub to exploit environment specific features. Our experiments show that space requirements and preprocessing times of MTSub are significantly lower than that of MtsCopa. Moreover, MTSub finds paths up to 29 times faster than an incremental algorithm, G-FRA*, and only up to 3.89 times slower than MtsCopa.

In the following sections, we define the problem and the notation, explain related work, discuss MTSub further with the results of the empirical analysis, and finally, share our conclusions and future work.



(a) Positions at time t_i (b) Positions at time t_{i+1}

Figure 1: a. Two possible paths, with the same cost, that the agent, A, can take to reach the target T at time t_i; b. Positions of A and T at time t_{i+1} after A chose one of the two paths randomly.

Problem Definition and Notation

In this section, we explain the notation used in this paper and formally define the problem. An undirected graph $G = (V, E)$ represents an 8-connected environment, where V denotes all the nodes in graph and E corresponds to allowable moves between neighbouring grid cells. The nodes can be either blocked or unblocked. The environment is static, which implies that nodes do not change their status over time, e.g., becoming an obstacle or ceasing being an obstacle. S denotes all the unblocked nodes in V, where $S \subset V$. Cardinal moves are allowed between adjacent nodes if both nodes are unblocked. Diagonal moves are allowed if both nodes are unblocked and the two corresponding cardinal nodes are also unblocked (here we assume the agent is as large as a grid cell). The costs of cardinal and diagonal moves are 1 and $\sqrt{2}$, respectively. We denote by $C(s, s')$ the smallest cost of traveling from s to s' where $(s, s') \in S$. We denote by $neighbour(s)$ the set of all nodes $s' \in S$ that s can move to either by an allowed diagonal move or by an allowed cardinal move. An ordered set of adjacent nodes, which form a path from s to s', is denoted by $\Pi(s, s')$, where $s, s' \in S$. We define $C(\Pi(s, s'))$ as cost of the path Π. We use t to indicate discrete time steps from the beginning of the search to determine a next move until the end of the search. Notations $position(A, t_i)$ and $position(T, t_i)$ are used to show locations of the agent and target at time t_i, respectively. For simplicity, we also use s_A and s_T to indicate the location of the agent and the target at the current time, respectively. At the beginning of a search, $s_A = position(A, t_0)$ and $s_T = position(T, t_0)$. We use $route$ to denote an ordered collection of $position(A, t)$, $t_0 \leq t \leq t_k$, where t_0 is the time that the agent started to move towards the target and t_k is the time that the agent reached the target.

The problem is defined as follows: Given a graph $G = (V, E)$, representing an 8-connected environment, an agent A and a moving target T, positioned at s_A and s_T respectively, find an optimal route that is subject to:

- A and T can move at discrete time steps.
- At each time step t_i, T positioned at $u = position(T, t_i)$ can either stay put or move to node v where $v \in neighbour(u)$.
- At each time step t_i, A, positioned at ($p = position(A, t_i)$), can detect the position of T ($u = position(T, t_i)$), compute a path $\Pi(p, u)$ and move to a node w where $w \in neighbour(u) \wedge w \in \Pi(p, u)$.

An optimal route is a route from $position(A, t_o)$ to $position(T, t_k)$, when t_o is the time that A and T attempted to move, t_k is the time that A reached T, and $\forall t_i\ t_o \leq t_i \leq t_k$, $position(A, ti) \in \Pi(position(A, t_{i-1}), position(T, t_{i-1}))$ where Π is a cost minimal path.

Note that following an optimal route may not minimize the cost of reaching the target. Figure 1 depicts a search environment at time t_i and t_{i+1} where black cells represent obstacles, 'A' and 'T' show s_A and s_T, respectively. Two obstacle free paths are displayed in Figure 1a. The paths have the same cost. However, Figure 1b shows that the agent's random choice of paths at time t_i may not be the best choice to get closer to the target. Experiments show that following an optimal route decreases the cost of reaching the target in most cases (Botea et al. 2013; Sun et al. 2012).

Please note that, an agent can evade indefinitely from a target with the same speed. This paper focuses on optimal movement strategy for an agent regardless of this fact.

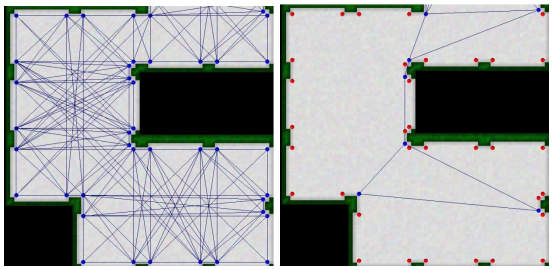
Background

This section summarizes algorithms and techniques that led to the Moving Target Search with Subgoal Graphs development. Algorithm A* and the Subgoal Graphs technique are discussed in order to provide more insight into the MTSub algorithm.

A* Algorithm

A* (Hart, Nilsson, and Raphael 1968) is a search algorithm that uses heuristics for calculating cost minimal paths between stationary locations. A* keeps track of four values for each node $s \in S$: 1. Approximation cost - $h(s)$ denotes user provided approximation of $C(s, s_T)$ where $s, s_T \in S$. The value $h(s)$ is computed by the function $H(s, s_T)$. If function H have the triangle inequality property ($H(s, s'') \leq H(s, s') + H(s', s'')$ for all $s, s', s'' \in S$), the h values are admissible (Koenig and Likhachev 2005). Function $H(s, s')$ calculates octile distances between s and s', which is the shortest distance between s and s' as if there were no obstacles. 2. Calculated value for $C(s_A, s)$ - this value is denoted $g(s)$. Initially, it is assigned a zero for $g(s_A)$ and infinity for each node $s \in S$ other than s_A before the search. 3. Estimated cost of reaching the target - this value is denoted by $f(s)$ where $f(s) = g(s) + h(s)$. 4. Parent of s - the value, $parent(s)$ denotes the node that is the parent of s in the search tree. Initially, $parent(u) =$ NULL.

A* also uses two lists: 1. $OPEN$ list - this list maintains the nodes that are going to be evaluated. Initially, it has only $position(A, t_0)$ as a member. 2. $CLOSED$ list - this list contains all the nodes that were evaluated (expanded). At every iteration of A*, a node s with minimum f value is extracted from $OPEN$ and added to $CLOSED$. Each node $s' \in neighbour(s)$ is evaluated as follows: if $g(s) + C(s, s') < g(s')$ then $g(s') = g(s) + C(s, s')$ and $parent(s') := s$. Node s' is then added to the $OPEN$ list if it is not already in $OPEN$. A* terminates and returns a cost minimal path if s_T is expanded. If the $OPEN$ list is empty and s_T is not expanded, A* terminates and reports that there is no obstacle free path between s_A and s_T).



(a) Local Subgoal Graph (b) Global Subgoal Graph

Figure 2: Basic and Two-Level Subgoals (Uras, Koenig, and Hernández 2013)

Subgoal Graphs

MTSub uses Subgoal Graphs (Uras, Koenig, and Hernández 2013) to create an abstraction of the search environment and to find a cost minimal path $\Pi(s_A, s_T)$. Variations of the technique produced the fastest optimal results in Grid-Based Path Planing Competition 2013 (Uras, Koenig, and Hernández 2013). Moreover, the required space to create Subgoal Graph was lower than any other competitors.

An algorithm based on Subgoal Graphs relies on environment abstraction to speed up the cost minimal path searches between stationary locations in the eight connected grid worlds. Instead of searching for a cost minimal path in a given eight connected graph, the method uses a subset of the graph which is referred to as Simple Subgoal Graph (SSG). It is faster to search in a Simple Subgoal Graph than in the original graph because the SSG has fewer number of edges and nodes. A Simple Subgoal Graph is depicted in Figure 2a.

Subgoal Graphs have some similarities to visibility graphs (Lozano-Pérez and Wesley 1979): 1. It is possible to add any node of the original graph to Subgoal Graph. This means it is possible to connect (with the function `connectstartandgoal` in (Uras, Koenig, and Hernández 2013)) s_A and s_T to SG at any time t. 2. The edge costs of Subgoal Graphs represent cost minimal paths between corresponding nodes. 3. If $C(s, s') > H(s, s')$, where s and s' are nodes of SG, there is at least one node u in SG that is on the cost minimal path $\Pi(s, s')$. 4. If s and s' are subgoals and there is an edge of SG that connects them, any $\Pi(s, s')$, where $C(\Pi(s, s'))= H(s, s')$, is a path. The second, third and fourth similarities imply that it is possible to find an ordered list of nodes of SG that is on the cost minimal path. Nodes of SG are called subgoals for this reason. This similarity means that it is trivial to transform a ordered list of the subgoals between s_A and s_T, which is referred to as a high level path, to $\Pi(s_A, s_T)$, which is referred to as a low level path.

The algorithm creates a SSG only once for a given environment. The generation of a SSG is as follows: An unblocked node s is marked as a subgoal if at least one of its diagonal neighbours is blocked and all of its cardinal neighbours are unblocked. An edge is created between two subgoals s and s' if and only if there is a path Π such as $C(\Pi(s, s')) = H(s, s')$ and there is no subgoal s'' such as $C(\Pi(s, s'')) + C(\Pi(s'', s')) = C(\Pi(s, s'))$ where $s' \in S$. If s and s' are connected in SG, $s \in sg.neighbour(s')$ and $s' \in sg.neighbour(s)$. Finding a high level path in SSG is done by setting s_A and s_T as subgoals and adding edges using the above rule. Once done, a high level path can be obtained using an A* search. Converting the high level path to a low level path is trivial.

Hierarchical abstraction can be used to speed up the search by creating layers of Subgoal Graphs in order to accelerate the search. Although, it is achievable to create a hierarchical abstract environment by using n-level subgoals (Uras and Koenig 2014), MTSub uses Two-Level Subgoal Graphs. Two-Level Subgoal Graph is depicted in Figures 2a and 2b. Here, level 1 is the SSG Figures 2a and level 2 is Figures 2b.

Nodes in the Two-Level Subgoal Graph, referred to as global subgoals, are a subset of the nodes of SSG, which is also referred to as local subgoals. The idea behind Two-Level Subgoals is to eliminate subgoals that are not contributing a cost minimal path between two SG neighbours that it connects.

Global subgoals are selected as follows: First, all of nodes in SSG are declared as global subgoals. Second, subgoals are checked for two properties: subgoal s is declared as local subgoal, where $\forall s\ s', s'' \in sg.neighbour(s)$ and $s, s', s'' \in S$, if 1. there is a high level path Π, excluding s, such as

$C(\Pi(s', s'')) \leq C(s', s) + C(s, s'')$ or 2. $C(\Pi(s', s'')) = H(s', s'')$. An edge (s', s'') is created if the second condition is satisfied but not the first.

A cost minimal path is found with Two-Level Subgoal Graphs as follows: First, s_A and s_T are added to the Global Subgoal Graph (GSG), if they are not already in GSG or SSG. Second, all SG neighbours u of s_A and s_T are declared as global subgoals where u is a local subgoal. Next, it is possible to find a high level path in the Two-Level Subgoal Graph using A*. A directed depth first search (searching with only corresponding cardinal and diagonal direction) is used to construct a low level path between two global subgoals s, s' because following corresponding cardinal and diagonal directions randomly from one global subgoal to another may not lead to a cost minimal path as it does in SSG. Therefore, transforming a high level path to a low level path in Two-Level Subgoal Graphs may be more costly than that of simple Subgoal Graphs.

Moving Target Search with Subgoal Graphs

Algorithm 1 Main Function and MTSub Algorithm

```
1: function Main
2:     if ¬ checkconnectivity(s_A, s_T) then
3:         return false;
4:     constructsubgoal(S);
5:     while s_A ≠ s_T do
6:         if the target moved then
7:             MTSub (S, s_A, s_T, Π);
8:         agentmove(Π);  \\agent follows the cost minimal path
9:         return true if s_A = s_T;
10:        targetmove();  \\target moves
11:    return true;
12: function MTSub(S, s_A, s_T, Π)
13:    if a search tree is constructed then
14:        prepareforsearch(S, s_T);
15:        expandsearchtree(S, s_T);
16:        if insubtree(S, s_A, Π) then
17:            return Π;
18:        else
19:            return initialsearch(S, s_A, s_T, Π);  \\build a new search tree
20:    else  \\if there is no usable search tree
21:        return initialsearch(S, s_A, s_T, Π);  \\build a new search tree
```

Moving Target Search with Subgoal Graphs(MTSub) is an optimal algorithm that attempts to update the path based on knowledge from the previous step. MTSub is different from the other incremental MTS algorithms because it uses a Two-Level Subgoal Graph, which is constructed during a preprocessing step.

MTS in known environments is a series of similar path planning problems. However, it is not possible to apply incremental search methods, that are used in grid graphs, directly to the Two-Level Subgoal Graphs. Because every location that an agent and a target can move, is represented with a node in grid graphs. In contrast, only a subset of the such locations are represented in Two-Level Subgoals. This implies that the current agent and target locations may not be a part of the Two-Level Subgoal. Therefore, the locations s_A, s_T and their local SG neighbours must be connected to the Two-Level Subgoal Graph before initiating a path search, $\Pi(s_A, s_T)$, at a time t. Unfortunately, inserting new nodes and edges to the Two-Level Subgoal Graph at every time t is time consuming (and the search is somewhat longer depending how many nodes and edges were added). In addition, those nodes must be removed after the search in order to preserve the Two-Level Subgoal Graph properties.

MTSub overcomes this problem as follows: First, if there is no tree data structure in SG (search tree), a new tree is constructed after s_A and s_T are connected to the Two-Level Subgoal Graph. After obtaining a path $\Pi(s_A, s_T)$, s_T and local SG neighbours of s_T are removed from the search tree and from the Two-Level Subgoal Graph (s_T is not added or removed if it is already a global subgoal). Note that this operation does not spoil the search tree connectivity, because the search tree stops expanding once it reaches s_T. Thus, there are no nodes in the search tree that are rooted at s_T. Local SG neighbours of s_T do not contribute to the cost minimal paths between two global nodes, by definition of global subgoals. Therefore, they do not connect two global nodes in the search tree and removing them does not violate connectivity of the search tree. Once there is an existing search tree, MTSub does not connect the current s_A to the Two-Level Subgoal but it connects s_T. Next, MTSub expands the search tree rooted at $position(A, t_i)$ where A is the agent and t_i is the time step when the search tree is initiated. After a cost minimal path $\Pi(position(A, t_i), s_T)$ is found, MTSub checks if s_A is on the path. If it is, MTSub returns a part of the path $\Pi(position(A, t_i), s_T)$ from s_A towards s_T. Note that the returned path is also a cost minimal path because it is a sub-path of a cost minimal path. If s_A is not on the path $\Pi(position(A, t_i), s_T)$, MTSub connects s_A to the Two-Level Subgoal Graph, constructs a new search tree that is rooted at s_A from scratch, and finds a path $\Pi(s_A, s_T)$. Note that the graph does not grow as we continue incremental search since we remove s_T at the end of every search. Details of the algorithm are explained below with the help of pseudo-code.

Algorithm `MTSub` checks if the agent and the target are positioned in the same cell of the the grid graph (line 2). If they are not in the same grid cell, then a path $\Pi(s_A, s_T)$ does not exist. Otherwise, a path $\Pi(position(A, t), position(T, t))$ at $\forall\ t$ exists. Next, the Two-Level Subgoal Graph is generated (line 4). The agent follows a cost minimal path constructed by MTSub until reaching the target. If the target moves, the path is updated with MTSub function (line 7).

MTSub is a function that returns a cost minimal path between s_A and s_T. MTSub executes as follows: If there is no existing search tree, MTSub calls `initialsearch` (line 21). The function uses A* to create a search tree over Two-Level Subgoal Graph and to find a cost minimal path. After the end of the function, a cost minimal path $\Pi(s_A, s_T)$ is generated and a reusable search tree rooted at s_A is initiated. The root of the tree, which is referred to as $position(A, t_i)$ where t_i is the time step when the search tree is built, stays

same until a new search tree is constructed.

Lemma 1. *The* `initialsearch` *function returns a cost minimal path* $\Pi(s_A, s_T)$.

Proof. The function uses an A* search over Two-Level Subgoal Graph to find a path. The heuristic function H that is used by the `initialsearch` obeys the triangle inequality. Octile distance used as heuristic and it gives the shortest possible distance between two nodes if we assume that there is no obstacles in the environment. Therefore, it is not possible to find a node s where $H(u, v) > H(u, s) + H(s, v)$ and $u, v, s \in S$. Thus, based on Theorem 2 in (Uras, Koenig, and Hernández 2013) an A* search over Two-Level Subgoal Graph returns a cost minimal path. □

Algorithm 2 Prepare For Search Function

```
22: function prepareforsearch(S, s_T)
23:     connecttarget(s_T); \\connect target location to the graph
24:     for all local subgoal s ∈ sg.neighbour(s_T) or node s = s_T do \\iterate through recently added nodes
25:         g(s) := ∞;
26:         for all node s' ∈ CLOSED and s' ∈ sg.neighbour(s) do
27:             if g(s) > g(s') + C(s', s) then
28:                 g(s):= g(s') + C(s', s);
29:                 parent(s):= s';
30:         if g(s) != ∞ then
31:             add s into OPEN;
```

Once MTSub has existing data, `prepareforsearch` function is executed (line 14) as a first step of the incremental search. The function makes required changes in Two-Level Subgoal Graph to the make existing tree usable with s_A and s_T. First, the function connects s_T to the graph with corresponding edges (line 23). Next, the function connects s_T and the local SG neighbours of s_T (nodes that are newly marked as global subgoals) to the search tree as follows: The function updates $parent$ accordingly (line 29) and the g value of every node s that was recently added to the Two-Level Subgoal Graph by minimizing $g(s) = g(s') + C(s', s)$, where $s, s' \in S$, $s' \in CLOSED$ and $s' \in sg.neighbour(s)$ (line 28). Node s is added to the $OPEN$ list, if $g(s) \neq \infty$ (line 31). MTSub calculates f values as $f(s) = g(s)+H(s, position(T, t_i))$ where $s \in S$ and $position(T, t_i)$ is the location of the target when the search tree is initiated. Note that heuristic values are not admissible after the execution of this function because the actual cost of s ($g(s)$) may be smaller than the provided heuristic value ($h(s)$).

Lemma 2. *After* `prepareforsearch` *function executes, the* $OPEN$ *list has all the required nodes for a cost minimal path search between the root of the search tree and* s_T.

Proof. Algorithm A* guarantees that if the $CLOSED$ list is not empty then $OPEN$ consists of nodes v such that $\forall\, v \in OPEN, \exists\, u \in CLOSED$ such that $u \in sg.neighbour(v)$. All the nodes in the $OPEN$ list has a parent in the $CLOSED$ list that minimizes $g(s) = g(s') + C(s, s')$ where s is a node in the $OPEN$ list and s' is a node in both $sg.neighbour(s)$ and $CLOSED$.

The function is invoked when a search tree is already in place and therefore the $OPEN$ and the $CLOSED$ lists are already populated. However, the $OPEN$ list may not be complete because the function adds s_T and every local subgoal s where $s \in sg.neighbour(s_T)$ to the graph after an A* search was executed.

To fix this problem, the function `prepareforsearch` examines every node s that was added to the Two-Level Subgoal Graph. Here the function either finds a node $s' \in CLOSED$ and $s' \in sg.neighbour(s)$ that minimizes $g(s') + h(s)$ or no such node s' exists. In the former case, the function sets $g(s) = g(s') + h(s)$ and $parent(s) = s'$ and it inserts s into the $OPEN$ list. In the latter case s is not reachable from any node in $s' \in CLOSED$ and therefore is not added to the $OPEN$ list.

Therefore, when `prepareforsearch` terminates, the $OPEN$ list has all required nodes for a search. □

Algorithm 3 Expand Search Tree Function

```
32: function expandsearchtree(S, s_T)
33:     while OPEN ≠ ∅ do
34:         extract a node s with the smallest g(s) + h(s) from OPEN;
35:         add s to CLOSED;
36:         if g(s) + h(s)> g(s_T)+H(s_T, position(T, t_i)) then
37:             construct path Π from a high level path and return ;
38:         for all s' that is sg.neighbour of s do
39:             if g(s') > g(s) + C(s, s') then
40:                 g(s'):=g(s) + C(s, s');
41:                 parent(s'):=s;
42:                 add s' into OPEN if it is not already in the list;
43:     return false;
```

Next, MTSub executes `expandsearchtree` to find a path between the root of the search tree and s_T (line 15). The function builds on the search tree by expanding nodes s until $f(s) < g(s_T) + H(s_T, position(T, t_i))$ where s is a node in the $OPEN$ list with the lowest f value, and $position(T, t_i)$ is the location of the target when the search tree is initiated (line 36). Note that we add $H(s_T, position(T, t_i))$ to $g(s_T)$ to make sure that every node that may be on the cost minimal path expanded even if they are disadvantaged because we do not use admissible heuristics in incremental search iterations.

Lemma 3. *The* `expandsearchtree` *function returns a cost minimal path.*

Proof. Algorithm A* does not end as long as there exists a node $s \in OPEN$ such as $g(s) + H(s, s_T) < g(s_T)$ in order to ensure that all possible paths leading to s_T were explored. Similarly, the function `expandsearchtree` does not terminate if there exists a node $s \in OPEN$ such as $g(s) + H(s, position(T, t_i)) < g(s_T) + H(s_T, position(T, t_i))$. This equation can be rewritten as $g(s) + H(s, position(T, t_i)) - H(s_T, position(T, t_i)) < g(s_T)$. By the triangle inequality, $H(s, position(T, t_i)) \leq H(s, s_T) + H(s_T, position(T, t_i))$. Therefore, the function does not terminate if there is a node s in

$OPEN$ such as $g(s) + H(s, s_T) < g(s_T)$. Hence, the `expandsearchtree` function returns a cost minimal path. □

After a path is constructed between the root of the tree and s_T, MTSub checks if s_A is on the path with the help of `insubgoal` function (line 16). If s_A is on the path, a cost minimal path $\Pi(s_A, s_T)$ is generated and returned. Otherwise, a new search tree is constructed with `initialsearch` function (line 19).

Lemma 4. *The `insubgoal` function returns a cost minimal path.*

Proof. The function `insubgoal` is only executed if there is a cost minimal path, $\Pi(position(A, t_i), s_T)$ where t_i is the time when search tree is built. The function returns a sub-path $\Pi(s_A, s_T)$ if both locations are on the path, $\Pi(position(A, t_i), s_T)$. Any sub-path between two nodes of a cost minimal path is also a cost minimal path. Therefore, the returned path is a cost minimal path. □

Theorem 1. *MTSub finds a cost minimal path* $\Pi(s_A, s_T)$.

Proof. MTSub returns paths that were computed by either `initialsearch` or `insubgoal` functions. All of these paths are cost minimal as shown in Lemmas 1, 3, and 4. Therefore, at each step MTSub returns cost minimal paths between current positions of the target and the agent. □

Theorem 2. *The position of an agent that is following a path generated by MTSub is on the optimal* $route$ *at any time* t_i.

Proof. According to Theorem 1, MTSub generates optimal paths between current locations of the agent and the target at $\forall\ t_i$. Thus, the position of the agent, $position(s_A, t_i)$, is on the cost minimal path $\Pi(position(s_A, t_{i-1}), position(s_T, t_{i-1}))$. Therefore, the agent is on the optimal $route$. □

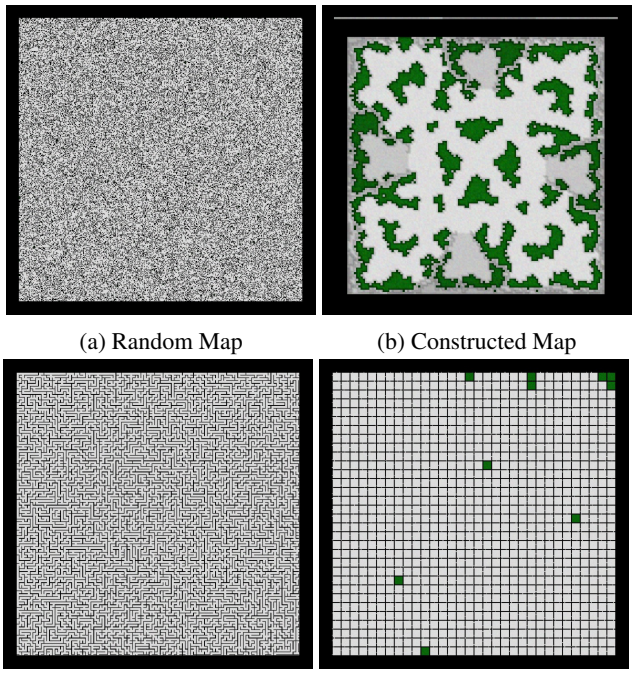
It is possible to use early termination techniques to find cost minimal paths to speed up incremental search.

The function `quickpath` which is introduced with Subgoal Graphs (Uras, Koenig, and Hernández 2013), checks if it is possible to find a path $\Pi(s_A, s_T)$ where $C(\Pi(s_A, s_T)) = H(s_A, s_T)$. The function tries to build such a path by starting from s_A and following the corresponding cardinal and diagonal nodes towards s_T. If an obstacle is encountered along the path, `quickpath` returns false. Otherwise, a cost minimal path is returned. We use this method at the beginning of the `MTSub` function to terminate early.

If there is an existing search tree, the `easypath` function is used to terminate early before the `prepareforsearch` function. This function checks if s_T is on the existing cost minimal path. If it is, we use the same path since any subpaths of a cost minimal path is a also cost minimal path.

Empirical Evaluation

In this section, we compare optimal algorithms for MTS performance.



(a) Random Map (b) Constructed Map

(c) Maze Map (d) Room Map

Figure 3: Map Classes

Methodology

We compare MTSub against two algorithms for MTS: G-FRA*, which is an incremental algorithm, and MtsCopa, which in the preprocessing phase computes a path between any two nodes $s, s' \in S$ and stores the results in a database. A previous paper (Sun et al. 2012) shows that the performance of FRA* and G-FRA* is similar and they have the smallest average run times among optimal algorithms that do not use preprocessing. Thus, we only compare G-FRA* against MTSub in the incremental algorithm category. MtsCopa has the smallest run times (Botea et al. 2013). We include MtsCopa in the experiments in order to demonstrate the efficiency of MTSub against an algorithm that exploits preprocessing. We also compared with algorithms for fixed location path planning, A* and Two-Level Subgoal Graphs, to demonstrate the benefits of incremental search. We use such algorithms repetitively to find path $\Pi(s_A, s_T)$ and solve MTS. We refer to them as Repeating A* (R-A*) and Repeating Two-Level Subgoal Graphs (R-Sub).

The experiments were conducted as follows: First, a predefined set of maps were selected. The size of a graph and obstacles were determined by a map. Second, scenarios were created which included initial locations of the agent and the target on the graph and target motion. The target path was defined by the scenario where the path moves were randomly selected (the path were stored and used by all algorithms). To hinder the target from running infinitely, we move the target at every time t_n, if its location at time t_{n-1} is not occupied by the agent at time t_n. The agent path was determined by the competing algorithms at run time. An experiment ends when s_A at time t_n is equal to s_T at time t_{n-1}

	Map AR0700SR		Map AR0500SR		Map AR0300SR	
	MTSub	MtsCopa	MTSub	MtsCopa	MTSub	MtsCopa
Preprocessing	0.18s	41299.2s	0.37s	12046s	0.5s	10489.2s
Memory	545KB	24064KB	501KB	7680KB	487KB	4403KB
Average	6.93μ	1.94μ	7.75μ	1.99μ	6.61μ	1.75μ
Maximum	184μ	27μ	149μ	84μ	147μ	25μ

Table 1: Comparison between MtsCopa and MTSub, with respect to preprocessing time and space, and performance. The Experiments were conducted on 320 x 320 Maps. Preprocessing shows the preprocessing time in seconds and memory shows the space requirements in KB after the preprocessing was completed. Average and Maximum are the average time per step and maximum time per step respectively.

and the target can not move.

We use four classes of maps to examine the effects of the environment on the different algorithms (Figure 3): 1. Constructed Maps - are used in well known video games such as "Dragon Age: Origins", "StarCraft", "Baldur's Gate" and "WarCraft". We use these maps to monitor performances of the algorithms in video games. 2. Random Maps - are created by adding blocked grid cells to random points on the map. These maps allows us to evaluate how algorithms behave if the environment is not suitable for abstraction. 3. Room Maps - have rooms that allow access other rooms through a small number of unblocked cells. We use these maps for observing performances if the environment is particularly good for abstraction. 4. Maze Maps - represent environments where user provided heuristics are usually misleading. These maps are used for examining the effects of heuristics on the algorithms. All maps used in the experiments are retrieved from Nathan Sturtevant's pathfinding repository (Sturtevant 2012).

We conduct the first set of experiments in maps scaled 512 x 512 with 100 scenarios per map. For each map class, we choose 6 maps[1]. MtsCopa was excluded from this set of experiments due to its significantly high preprocessing time and memory footprint. In the second set of experiments, we use three smaller, 320x320, constructed maps with 100 scenarios for each map to compare MTSub and MtsCopa. We also report preprocessing requirements of MTSub for some instances of Constructed Maps ranging from 512 x 512 to 1024 x 1024.

All of the algorithms are implemented comparatively, using original authors' codes. We run experiments on a 2.50 GHz Linux machine with 6 GByte of RAM. All of the reported times are CPU time. We ensured that algorithms did not require the use of virtual memory and thus access the disk repeatedly.

We measured required time and space (memory) for preprocessing to discuss applicability of the algorithms that exploit preprocessing. We report average time spent to determine a next move at a time step t to give a notion of run time performance. We also report maximum time spent per step to see if algorithms meet the real time requirements. We do not report the cost of reaching the target because all of the

[1]Constructed Maps: AR0011SR, AR0071SR, AR00511SR, stormguarde, duskwood, tranquilpaths; Random Maps: random512-20-1, random512-20-2, random512-20-3, random512-20-4, random512-20-5, random512-20-6; Maze Maps: maze512-2-1, maze512-2-2, maze512-2-3, maze512-2-4, maze512-2-5, maze512-2-6; Room Maps: 16room_001, 16room_002, 16room_003, 16room_004, 16room_005, 16room_006

		R-A*	R-Sub	GFRA*	MTSub
Constructed	Avg.	325.20	12.09	140.54	6.85
	Max	9369	188	26857	144
Random	Avg	284.2	270.10	91.66	50.56
	Max	5389	6588	15533	5140
Room	Avg.	626.48	23.09	163.25	5.66
	Max	9858	810	33507	304
Maze	Avg.	8571.96	852.71	648.5	44.85
	Max	38315	7871	80054	4523

Table 2: Experiments in 512 x 512 Maps, Times are Given in μ seconds. All reported times are per step where Avg. is the average time per step and Max is the maximum time per step. Note the effect of map type on the performance of the algorithms.

algorithms find optimal solutions according to the information available to the agent and the results are very close. Note that, MtsCopa is an optimal algorithm for MTS because it always finds the first step of a cost minimal path, $\Pi(s_A, s_T)$ at $\forall t$.

Experiments and Results

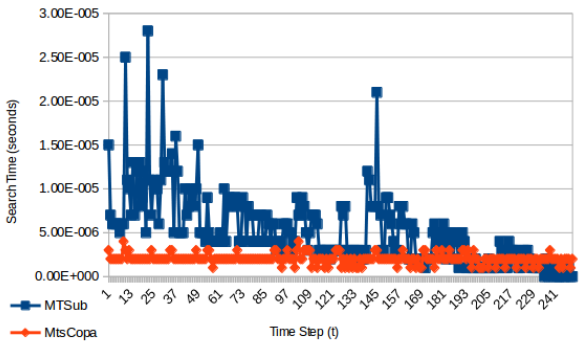
Results of the first set of experiments are depicted in Table 2.

One of the incremental algorithms, G-FRA* has smaller average runtime than that of R-A* but R-A* has smaller maximum time than that of G-FRA* in all of the map domains. These results indicate that repairing a search tree during runtime, increases maximum time spent in a search step even if it decreases average runtime. Please note that, MTSub does not have the same issue and it outperforms R-Sub in both average and maximum run times. Because MTSub only expands an existing search tree or builds a new search tree from scratch as needed.

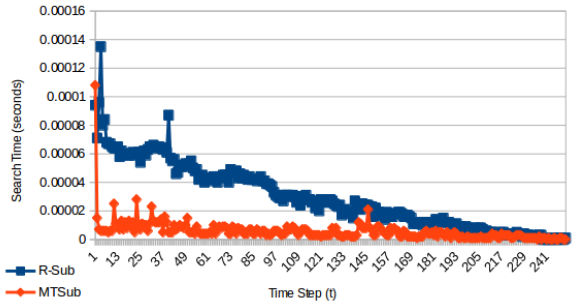
R-Sub outperforms A* in both maximum and average times in all of the map classes except Random Maps. The environment abstraction technique used by R-Sub can not generate significantly smaller abstract graphs for Random Maps. Therefore, R-Sub can not outperform A* remarkably in such environments. This indicates that environment abstraction can be used to decrease not only average times but also maximum times when the technique can be applied to the environment effectively.

The incremental algorithm G-FRA* has smaller average times than that of R-Sub not only in Random Maps but also in Maze Maps. This implies that incremental search is comparatively more effective in Maze Maps and Room Maps because in often times, the target does not move to locations that make G-FRA* repair its search tree.

The suggested algorithm, MTSub outperforms all of the competing algorithms in both average and maximum times in this set of the experiments. MTSub uses both environment



(a) MTSub vs. MtsCopa



(b) MTSub vs. R-Sub

Figure 4: Run Times per Search in Map AR0700SR

abstraction and incremental search and successfully avoids the disadvantages of those techniques. The results also show that the run time performance of MTSub is closely related combined success of the techniques it uses.

The results of the second set of experiments, shown in Table 1, depict preprocessing time and space consumption in addition to average and maximum search times.

MTSub outperforms MtsCopa up to 229400 times in preprocessing time and up to 44.15 times in space requirements. Nonetheless, MTSub is slower than MTSCopa but it is competitive in average and maximum runtime per search. MtsCopa is better than MTSub by only up to 3.89 times in average search times and up to 6.81 times in maximum times.

MTSCopa has consistent step time requirements across the map classes according to our experiments and analysis of the data from GPPC[2]. The expected MTSub performance for random maps will not be as good as in the case of Constructed/Room map. The performance degradation is a result of the large number of random obstacles, which impacts the size of the subgoal graph. Here we expect the MTSub performance to be about 20 times slower than MTSCopa, which is more significant.

We give required preprocessing time and space for MTSub in various maps in Table 3 to complement results of the second set of experiments. The results show that MTSub is applicable to large maps.

We also examined search times individually in a Constructed Map AR0700SR to complement the experimental results. Figure 4 shows run times per search of MTSub, MtsCopa and R-Sub in a scenario for AR0700SR map. Search times are given per time t from $t = 1$ to $t = 250$. All of the agents reach the target at $t = 250$. In figure 4a, we compare MTSub against MtsCopa. Compared algorithms spend maximum search time at the beginning which is 108 μs for MTSub and 11 μs for MtsCopa. We exclude maximum times from this graph to better depict the change in search times in runtime. The graph shows that the runtime of MTSub has jumps. The jumps occur when the existing search tree is no longer usable and MTSub constructs a search tree from scratch. These jumps do not exceed the maximum runtime and get smaller as the agent gets closer to the target. However, MtsCopa maintains almost the same runtime after the maximum time without affecting from the distance between agent and the target. This is because MtsCopa finds the only next move on a cost minimal path without returning a complete path. In Figure 4b, we compare MTSub and R-Sub. We did not exclude maximum search times in this graph. It is shown that, after the first time step, MTSub finds paths faster than a repeating algorithm that also uses Two-Level Subgoal Graphs for search. This implies that incremental search effectively decreases the response time of the agent.

Map Name	Map Size	Preprocessing Time	Space
Aftershock	512 x 512	1989.26ms	1.3MB
Backwoods	512 x 768	754.164ms	1.9MB
RiverLethe	512 x 768	1685.51ms	1.9MB
Turbo	768 x 768	1691.91ms	2.6MB
Nightshade	768 x 1024	4348.23ms	3.7MB
TheFrozenSea	1024 x 1024	7691.85ms	5.4MB
Octopus	1024 x 1024	5571.44ms	4.9MB
Cauldron	1024 x 1024	5063.05ms	5.2MB

Table 3: Preprocessing Requirements of MTSub

Conclusions and Future Work

In this paper, we introduced an innovative incremental search algorithm that uses environment abstraction to speed up the response time of an agent that searches for a moving target. The MTSub algorithm, finds optimal route between the agent and the target by computing cost minimal paths between the agent and the target at each time step. Empirical evaluation shows that MTSub outperforms G-FRA*, R-Sub and R-A* in all of the experiment domains. It is also demonstrated, using empirical analysis, that MTSub has a significantly smaller memory footprint and preprocessing time than MtsCopa, a state-of-art algorithm for Moving Target Search and that MTSub is only up to four times slower than MtsCopa in average run times. MTSub average performance meets real time requirements and is applicable to video games and robotics that use not only small environments but also larger ones with lower resources.

Acknowledgments

We wish to thank the anonymous reviewers for their constructive comments.

[2]http://movingai.com/GPPC/results.html

References

Botea, A.; Baier, J. A.; Harabor, D.; and Hernández, C. 2013. Moving target search with compressed path databases. In *ICAPS*, 288–292.

Hahn, G., and MacGillivray, G. 2006. A note on k-cop, l-robber games on graphs. *Discrete mathematics* 306(19):2492–2497.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on* 4(2):100–107.

Ishida, T., and Korf, R. E. 1991. Moving target search. In *IJCAI*, volume 91, 204–210.

Ishida, T., and Korf, R. E. 1995. Moving-target search: A real-time search for changing goals. *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 17(6):609–619.

Koenig, S., and Likhachev, M. 2005. Fast replanning for navigation in unknown terrain. *Robotics, IEEE Transactions on* 21(3):354–363.

Lozano-Pérez, T., and Wesley, M. A. 1979. An algorithm for planning collision-free paths among polyhedral obstacles. *Communications of the ACM* 22(10):560–570.

Moldenhauer, C., and Sturtevant, N. R. 2009. Optimal solutions for moving target search. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems-Volume 2*, 1249–1250. International Foundation for Autonomous Agents and Multiagent Systems.

Strasser, B.; Harabor, D.; and Botea, A. 2014. Fast first-move queries through run-length encoding. In *Seventh Annual Symposium on Combinatorial Search*, 157–165.

Sturtevant, N. R. 2012. Benchmarks for grid-based pathfinding. *Computational Intelligence and AI in Games, IEEE Transactions on* 4(2):144–148.

Sun, X.; Yeoh, W.; Uras, T.; and Koenig, S. 2012. Incremental ara*: An incremental anytime search algorithm for moving-target search. In *Twenty-Second International Conference on Automated Planning and Scheduling*, 243–232.

Sun, X.; Yeoh, W.; and Koenig, S. 2009. Efficient incremental search for moving target search. In *Twenty-First International Joint Conference on Artificial Intelligence*, 615–620.

Sun, X.; Yeoh, W.; and Koenig, S. 2010. Moving target d* lite. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 1-Volume 1*, 67–74. International Foundation for Autonomous Agents and Multiagent Systems.

Uras, T., and Koenig, S. 2014. Identifying hierarchies for fast optimal search. In *Seventh Annual Symposium on Combinatorial Search*, 878–884.

Uras, T.; Koenig, S.; and Hernández, C. 2013. Subgoal graphs for optimal pathfinding in eight-neighbor grids. In *Twenty-Third International Conference on Automated Planning and Scheduling*, 224–232.

Vieira, M. A.; Govindan, R.; and Sukhatme, G. S. 2008. Optimal policy in discrete pursuit-evasion games. *Department of Computer Science, University of Southern California, Tech. Rep* 08–900.

Vieira, M. A.; Govindan, R.; and Sukhatme, G. S. 2009. Scalable and practical pursuit-evasion with networked robots. *Intelligent Service Robotics* 2(4):247–263.