Dynamically Pruned A* for Re-planning in Navigation Meshes

Wouter van Toll and Roland Geraerts
Dept. of Information and Computing Sciences, Utrecht University
Princetonplein 5, 3584 CC Utrecht, The Netherlands

***Abstract*— Modern simulations feature crowds of AI-controlled agents moving through dynamic environments, with obstacles appearing or disappearing at run-time. A dynamic navigation mesh can represent the traversable space of such environments. The A* algorithm computes optimal paths through the dual graph of this mesh. When an obstacle is inserted or deleted, the mesh changes and agents should re-plan their paths. Many existing re-planning algorithms are too memory-intensive for crowds, or they cannot easily be used on graphs where vertices and edges are added or removed.**

In this paper, we present *Dynamically Pruned A (DPA*), an extension of A* for re-planning optimal paths in dynamic navigation meshes. DPA* has similarities to adaptive algorithms that make the A* heuristic more informed based on previous queries. However, DPA* prunes the search using *only* the previous path and its relation to the dynamic event. We describe the four re-planning *scenarios* that can occur; DPA* uses different rules in each scenario. Our algorithm is memory-friendly and robust against structural changes in the graph, which makes it suitable for crowds in dynamic navigation meshes. Experiments show that DPA* performs particularly well in complex environments and when the dynamic event is visible to the agent. We integrate the algorithm into crowd simulation software to model large crowds in dynamic environments in real-time.**

I. Introduction

In modern simulations, crowds of virtual characters (or *agents*) must plan and traverse paths through complicated environments in real-time. Agents should move smoothly and avoid collisions with obstacles and other agents. A common approach is to let a *navigation mesh* represent the areas in which the agents can move. The *dual graph* of this mesh has a vertex for each mesh polygon and an edge for each pair of adjacent polygons. Agents use the A* search algorithm [1] on this graph to find *global* routes, which they traverse while *locally* avoiding other agents.

In *dynamic environments*, obstacles can appear, disappear, or move during the simulation. When the effect is sufficiently large, a local collision avoidance method may not be able to guide agents towards their goals: an agent may get stuck along its old route, instead of looking for a detour. For example, imagine a bridge collapsing, an explosion opening up a new route, or a large vehicle blocking an alley.

When such a dynamic event occurs, the navigation mesh should be updated and agents should *re-plan* their global paths. Efficient re-planning algorithms exist for graphs with dynamic costs and for high-dimensional motion planning problems. However, many of these algorithms require too much memory for crowds (because agents need to remember parts of the previous search), or they are difficult to implement for graphs in which vertices and edges (dis)appear.

In this paper, we present *Dynamically Pruned A** (DPA*), an extension of A* that efficiently re-plans an optimal global path when an obstacle has been inserted or removed. DPA* *prunes* the search based only on the agent's previous path and its relation to the event, which can always be described using one of four *scenarios*. Conceptually, DPA* is closest to *adaptive* A* techniques [2] which improve the A* heuristics based on memory of the previous search. However, DPA* prunes the search based only the previous path and the re-planning scenario; hence, it is tailored for applications with limited memory per agent. We show that DPA* outperforms A* in large graphs, especially when the dynamic event is visible to the agent. DPA* can be used for large real-time crowds in dynamic environments.

II. Related Work and Preliminaries

A. Crowd Simulation in Static Environments

In many crowd simulation systems, an agent finds a global route through the environment by performing an A* search on a grid or (more generally) a graph. Graphs are commonly used for high-dimensional motion planning [3], [4] where paths are often smoothed afterwards. They are less suitable for crowds on walkable surfaces: agents would need to either follow the edges exactly (which may lead to collisions between agents) or perform expensive geometric tests to check how they can deviate from an edge. Grids subdivide the environment into regular cells. They are intuitive and easy to implement [5]. However, grids have resolution problems: a coarse grid does not capture the environment's details, whereas a fine grid is expensive to store and query.

By contrast, navigation *meshes* efficiently subdivide the walkable space into polygonal regions. Using a mesh, an agent finds a sequence of regions to move through, and it can use the surrounding free space to locally adjust its movement. Many navigation meshes exist for 2D environments [6]–[8] and for multi-layered 3D environments [9]–[12].

Global planning on a navigation mesh leads to an *indicative route* that the agent can traverse smoothly in real-time [13] while locally avoiding other agents [14]–[18]. Hence, the agents themselves are typically *not* modelled as navigation mesh obstacles. Local avoidance is outside the scope of this paper, but it is worth noting that local avoidance is not sufficient when the global path needs to change.

B. A* search

The A* algorithm [1] finds a path through a graph from a start vertex S to a goal vertex G by performing *best-first*

search. Starting at S, A* iteratively expands the vertex V for which the sum $g(V)+h(V)$ is lowest. Here, $g(V)$ is the *cost* of the best discovered path from S to V so far, and $h(V)$ is a *heuristic* that estimates the cost of the optimal path from V to G. The vertices to explore are stored in an *open list*, sorted by their values of $g+h$. Vertices that have already been expanded can be stored in a *closed list*. If h is *admissible* (i.e. it never overestimates the optimal path cost), then A* computes an optimal path. If h is also *consistent* (i.e. the decrease in h is never larger than the increase in g), then the vertices in the closed list never need to be revisited. Costs and heuristics are often distance-based, but not necessarily.

Since our DPA* algorithm *prunes* the standard A* search without changing any costs or heuristics, it does not affect admissibility or consistency. When an admissible heuristic is used, DPA* computes optimal paths.

C. Dynamic Environments

When a large or complex obstacle appears or disappears dynamically, local collision avoidance is often not sufficient to let an agent reach its goal. Instead, the environment representation should be updated, and agents should re-plan their global paths (e.g. to find a detour).

In theory, a time dimension can be added to represent changes in the environment [4], but this is impractical for large crowds in real-time. Early approaches for crowds in dynamic environments were based on adaptive graphs [19], [20]. Modern navigation meshes allow efficient on-line insertions and deletions of obstacles [21], [22].

D. Re-planning Algorithms

To efficiently re-plan optimal paths after a dynamic event, *incremental* variants of A* deal with changing costs by remembering information from the previous query. Algorithms such as *D* Lite* and *Fringe-Saving A** remember the g and h values of each graph vertex and update the values that change due to the event [23]–[26]. These are related to *anytime* algorithms that iteratively improve a sub-optimal path [27]–[29]. However, remembering the A* search space of each agent is not feasible for large crowds. Also, re-planning in a dynamic navigation mesh is more complex than in a graph in which the costs change but the structure does not. A dynamic event may cause parts of the dual graph to (dis)appear; we cannot simply apply different costs to graph edges that already existed. Handling these effects in a memory-based algorithm is possible in theory, but difficult and costly in practice. For these reasons, DPA* does not require memory of the previous search other than the path itself.

Another approach is to use *experience graphs* [30], in which only an abstract higher-level graph is remembered. This is particularly useful for high-dimensional motion planning problems; it is less applicable to our problem, since navigation meshes are already compact.

E. Re-planning: Adaptive A*

The *Adaptive A** algorithm and its successors are closest to our work: they make the h values of vertices more informed by using the previous query, in such a way that h remains consistent [2], [31]–[33]. Under certain conditions, the algorithms can immediately stop when a vertex of the old path is expanded. These algorithms require less memory of the previous search than e.g. D* Lite, and they are more suitable for dynamic navigation meshes.

By contrast, DPA* does not make h more informed; instead, it uses the estimated 'distance' to the dynamic event to find out if vertices can be skipped. Hence, it *prunes* the A* search without changing any costs or heuristics. Our algorithm does not terminate until the goal vertex is expanded, which might be seen as a disadvantage compared to adaptive A*. However, in exchange, DPA* uses even less memory: agents only need to remember their paths. Furthermore, DPA* uses a distinction between four *scenarios* to optimize the search based on the situation at hand.

In short, DPA* investigates how some parts of the A* search can be skipped after a dynamic event, without requiring any extra data structures between and during queries. Conceptually, the method lies between regular A* and adaptive re-planning algorithms. DPA* is highly effective for real-time crowd simulation in dynamic environments.

III. Problem Description

A. Navigation Mesh and Graph

In this work, we assume that the agents use a *graph* for global planning. A 2D environment should have a planar graph describing the walkable space. A multi-layered environment should have multiple planar graphs (one for each layer) that are connected accordingly. For a navigation mesh, we can easily obtain a *dual graph* by assigning a vertex to each mesh region, and connecting the vertices of adjacent regions. Any path through this graph can be converted to a short and smooth path using the underlying mesh regions.

We also ensure that agents always plan paths between two *vertices* of the graph. This can be achieved by connecting the start and goal positions to nearby vertices. To allow this, the mesh should have a *point location* data structure that computes the region in which a query point is located.

B. Dynamic Events

In a dynamic environment, obstacles can be inserted, deleted, or moved during the simulation. We focus on *insertions* and *deletions*. Moving obstacles can be represented by sequential deletions and insertions; also, they are often treated as locally avoidable entities until they become stationary. Such a *dynamic event* leads to an update of the navigation mesh: regions can be added, removed, split, or merged. Consequently, the structure of the dual graph also changes. As mentioned, this means that we cannot easily use re-planning algorithms designed for graphs in which only the costs are dynamic and the topology is static.

Intuitively, when an obstacle is *inserted*, the area around it becomes more costly (or even impossible) to traverse; when an obstacle is *removed*, its neighborhood becomes easier to traverse. Let the *affected region* $\mathcal{R}$ be the part of the graph that has changed, i.e. the set of vertices and edges

that have appeared or disappeared. DPA* will treat $\mathcal{R}$ as an area in which the *costs* have increased (due to an insertion) or decreased (due to a deletion), regardless of what this area looked like before the event. We do not refer to individual edge costs in $\mathcal{R}$, but to the overall cost of passing through this area. Note that $\mathcal{R}$ is computed during the mesh update; we do not need to find it afterwards. Also, $\mathcal{R}$ can have any shape; it may even consist of multiple disconnected regions.

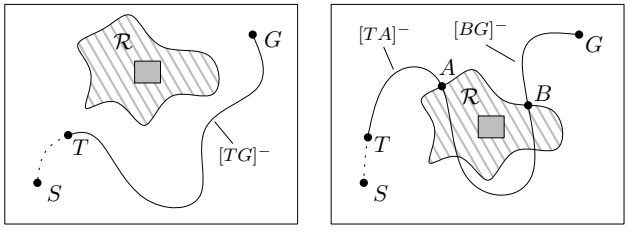
C. Re-planning Scenarios

A dynamic event can change the optimal path for an agent in a number of ways. This section describes which scenarios can occur; DPA* will use different rules for each scenario.

Let S and G be the start and goal vertex of an agent. Initially, the agent uses A* to find an optimal path in the graph, which we call $[SG]^-$. The superscript $^-$ refers to *old* paths, computed *before* a dynamic event. Assume that an event occurs later in the simulation, and the agent decides to re-plan when it has traversed the path up to a vertex T, e.g. because it can now see the event. The agent should re-evaluate its path from T to G, i.e. $[TG]^-$. We assume that the affected region $\mathcal{R}$ (the set of vertices and edges with updated costs) was already computed during the mesh update.

There are now two options: $[TG]^-$ either does or does not run through $\mathcal{R}$. If $[TG]^-$ does *not* run through $\mathcal{R}$ (Figure 1a), then the path is still valid, but it may not be optimal anymore. If $[TG]^-$ *does* run through $\mathcal{R}$ (Figure 1b), then it can enter and exit $\mathcal{R}$ multiple times, because $\mathcal{R}$ and the old path can be arbitrarily shaped. In the latter case, let A and B be the first and last vertex in $\mathcal{R}$ that occur in $[TG]^-$. We split the path into three sections: two *valid* subsections $[TA]^-$ and $[BG]^-$, and one *invalid* subsection $[AB]^-$. Note that A and T can be the same vertex, as well as B and G; hence, the valid subsections can be empty.

Furthermore, the event can be either an *insertion* or a *deletion* of an obstacle. This leads to four possible scenarios. We will now define DPA* for each scenario.



(a) Unaffected path (b) Affected path

Fig. 1: Re-planning scenarios after a dynamic event. An agent is following a path from S to G, and decides to re-plan at a vertex T. The inserted or deleted obstacle is shown as a gray rectangle, surrounded by the affected graph region $\mathcal{R}$. (a) If the old path $[TG]^-$ does not run through $\mathcal{R}$, then it is still valid, but possibly not optimal. (b) Otherwise, we define A and B as the first and last path vertex intersecting $\mathcal{R}$. The information in $[TA]^-$ and $[BG]^-$ is reused by DPA*.

IV. Dynamically Pruned A*

In all scenarios, a straightforward way to compute a new optimal path $[TG]^+$ is to perform A* from scratch. The superscript $^+$ refers to *new* paths, i.e. paths computed *after* the dynamic event. We now present Dynamically Pruned A* (DPA*), which adds *pruning rules* to the standard A* algorithm, by reusing information from the old path.

DPA* has different details in each re-planning scenario. It can be applied to all graphs with non-negative edge costs, including the dual graphs of navigation meshes. In contrast to many other re-planning algorithms, we do not require extra memory of the search space throughout the simulation; all information can be computed on the fly.

Scenario 1: Insertion, old path unaffected. If an obstacle has been inserted and $[TG]^-$ does *not* run through $\mathcal{R}$, then $[TG]^-$ is still optimal. Because the graph costs have not changed outside of $\mathcal{R}$, the old path $[TG]^-$ is still optimal among all possible paths that do not involve $\mathcal{R}$. Within $\mathcal{R}$, the costs have only increased, so any new path through $\mathcal{R}$ will be at least as expensive as $[TG]^-$. Hence, the agent does not need to re-plan, and DPA* simply returns $[TG]^-$.

Scenario 2: Insertion, old path affected. If an obstacle has been inserted and $[TG]^-$ *does* run through $\mathcal{R}$, then the agent may need to take a long detour around $\mathcal{R}$. In many cases, though, parts of $[TG]^-$ can be reused.

In Figure 1b, we observe that for any vertex $C \in [BG]^-$ (ignoring G itself), the optimal path from C to G will still be $[CG]^-$; it cannot have changed due to the increased costs in $\mathcal{R}$. We do not know in advance when the search will explore such a vertex C, but when it does, we know what the rest of the path via C should look like. In particular, we know that G is optimally reached from C via the *successor* of C in $[BG]^-$. When DPA* expands C, it adds only this successor to the open list, and not the other adjacent vertices. This prevents the algorithm from re-exploring some unnecessary branches. We emphasize that the search does *not* yet terminate at C, unlike in variants of Adaptive A* [2]. At this point, we have found an optimal path *via* C, but there may be an even better path that meets $[BG]^-$ at a later vertex. Thus, the search continues, but we ignore all paths via C that will definitely not be better.

Likewise, for each vertex $D \in [TA]^-$, we know that $[TD]^-$ will remain the optimal path from T to D. Specifically, D can optimally be reached from T via the *predecessor* of D in $[TA]^-$. Thus, when DPA* expands a vertex V with an adjacent vertex $D \in [TA]^-$, it only adds D to the open list if V is the predecessor of D. This prevents unnecessary updates of the open list. When using inconsistent heuristics, it also prevents D from being expanded when we know it will have a better parent vertex later on.

Algorithm 1 gives the pseudocode of DPA* for Scenarios 1 and 2 combined. For simplicity, the pseudocode does not include a 'closed list' of expanded vertices. When using admissible heuristics, a closed list can be added in the same way as in standard A* to further speed up the algorithm.

Algorithm 1 DPA*-INSERTION($T, G, [SG]^-, \mathcal{R}$)

if $[TG]^-$ does not pass through $\mathcal{R}$
  return $[TG]^-$
Determine $[TA]^-$ and $[BG]^-$
$g(T) \leftarrow 0$, T.*parent* $\leftarrow$ NULL
OPEN $\leftarrow \{T\}$
while *OPEN* $\neq \emptyset$
  $V \leftarrow \text{argmin}_{V' \in \textit{OPEN}}\{g(V') + h(V')\}$
  Remove V from *OPEN*
  if $V = G$
    return the path from T to G via parent pointers
  for each outgoing edge (V, V')
    if $V \in [BG]^-$ **and** $V' \neq$ *succ*$(V, [BG]^-)$
      continue
    if $V' \in [TA]^-$ **and** $V \neq$ *pred*$(V', [TA]^-)$
      continue
    if $g(V) + c(V, V') < g(V')$
      $g(V') \leftarrow g(V) + c(V, V')$
      V'.*parent* $\leftarrow V$
      Insert or update V' in *OPEN*
return NULL

Scenario 3: Deletion, old path unaffected. If an obstacle has been deleted and $[TG]^-$ does *not* run through $\mathcal{R}$, then $[TG]^-$ may contain a detour around an area that has now become more attractive. In general, a new optimal path may enter and exit $\mathcal{R}$ multiple times; we cannot know in advance when this will happen. However, $\mathcal{R}$ is the *only* region in which the costs have decreased. Thus, *if* there is a better path than $[TG]^-$, it *must* pass through $\mathcal{R}$ at least once.

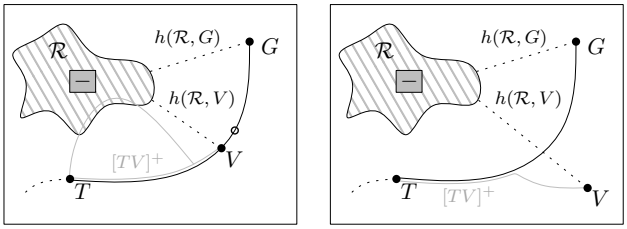
For this scenario, DPA* recognizes vertices for which a better path via $\mathcal{R}$ cannot exist. Let $c^*(V, \mathcal{R})$ be the (currently unknown) optimal path cost from a vertex V to any vertex in $\mathcal{R}$. Let $h(V, \mathcal{R})$ be a heuristic that does not overestimate this cost. For example, when using distance-based costs, $h(V, \mathcal{R})$ could be the Euclidean distance from V to a bounding polygon of $\mathcal{R}$. Note that $h(V, \mathcal{R}) = 0$ if $V \in \mathcal{R}$.

When expanding a vertex V, the costs for reaching the goal from V via $\mathcal{R}$ will be at least $h'(V) = h(V, \mathcal{R}) + h(G, \mathcal{R})$. The cost of an optimal path that (re-)visits $\mathcal{R}$ after V will be at least $g(V) + h'(V)$. If this value is greater than or equal to the cost of $[TG]^-$, then there is no point in visiting $\mathcal{R}$ from V, and we say that V is $\mathcal{R}$-*worse*. (Intuitively, if distance-based costs are used, we could say that '$\mathcal{R}$ is too far away'.) Note that all vertices explored from an $\mathcal{R}$-worse vertex will also be $\mathcal{R}$-worse themselves.

When DPA* arrives at a vertex V that is $\mathcal{R}$-worse, there are two cases in which the search can be pruned:

- If $V \in [TG]^-$, then the old subpath $[VG]^-$ is still optimal, because all paths via $\mathcal{R}$ are too costly. The best option is to follow the old path, so DPA* adds only the successor of V to the open list. It does not matter whether $[TV]^+$ has already visited $\mathcal{R}$. (See Figure 2a.) Again, as in Scenario 2, note that the search *does not terminate yet*, because there could still be better paths that meet $[TG]^-$ at a later vertex.
- If $V \notin [TG]^-$ and $[TV]^+$ has not passed through $\mathcal{R}$ yet, then there is no better path via V at all. After all, $\mathcal{R}$ must be visited at least once, and it is impossible to reach $\mathcal{R}$ from V and still obtain an optimal path. Hence, DPA* does not expand V any further. (See Figure 2b.)

This way, the open list contains only the vertices of $[TG]^-$, plus the vertices for which a better path through $\mathcal{R}$ might exist. As such, DPA* is still guaranteed to find an optimal path to G, either via the old path or via $\mathcal{R}$.



(a) Vertex on the old path (b) Vertex not on the old path

Fig. 2: Re-planning due to a deletion. DPA* has arrived at a vertex V for which a better path via $\mathcal{R}$ cannot exist, using an estimate of the distance to $\mathcal{R}$ (dotted segments). (a) If $V \in [TG]^-$, then the best path to the goal is still $[VG]^-$. DPA* adds only V's successor (black circle) to the open list. (b) If $V \notin [TG]^-$ and the new path $[TV]^+$ has not visited $\mathcal{R}$ yet, then there is no need to expand V.

Scenario 4: Deletion, old path affected. If an obstacle is deleted and $[TG]^-$ *does* run through $\mathcal{R}$, then $[TG]^-$ passes through affected navigation mesh regions. The *geometric* path is still obstacle-free, but it can possibly be improved. DPA* solves this similarly to Scenario 3: the new path should pass through $\mathcal{R}$ at least once and it cannot have a higher cost than $[TG]^-$. The difference is that only the subpath $[BG]^-$ still exists and may be re-used. Hence, instead of checking whether $V \in [TG]^-$, we now check whether $V \in [BG]^-$.

Algorithm 2 gives the pseudocode of DPA* for Scenarios 3 and 4 combined. To improve efficiency, we have added case distinction when checking which edges of V to explore (i.e. all neighbours, only its successor, or none). This postpones the check for $\mathcal{R}$-worseness until it can actually lead to pruning. Again, we omit the closed list for convenience.

V. EXPERIMENTS AND RESULTS

We have implemented DPA* for the dynamic Explicit Corridor Map (ECM) navigation mesh [7], [10]. The ECM is the *medial axis* [34] of the free space annotated with nearest-obstacle information. Path planning is performed on this medial axis, which is a sparse graph with O(n) vertices and edges, where n is the number of obstacle vertices. It supports on-line insertions and deletions of convex polygonal obstacles [22], which affect the ECM only locally.

We use Euclidean distance-based costs and heuristics. For deletions, we estimate the cost to $\mathcal{R}$ by the distance to the axis-aligned bounding box of all affected vertices. We include a closed list in both DPA* and regular A*, which is safe because the heuristics are admissible.

The software was written in C++ in Visual Studio 2013, and run on a Windows 7 PC with a 3.20 GHz Intel i7-3930K

Algorithm 2 DPA*-DELETION($T, G, [SG]^-, \mathcal{R}$)

if $[SG]^-$ passes through $\mathcal{R}$
  Determine $[TA]^-$ and $[BG]^-$
else
  $[BG]^- \leftarrow [TG]^-$
$g(T) \leftarrow 0$, T.*parent* $\leftarrow$ NULL
T.*visitedR* $\leftarrow$ **false**, T.*rworse* $\leftarrow$ **false**
OPEN $\leftarrow \{T\}$
while *OPEN* $\neq \emptyset$
  $V \leftarrow \text{argmin}_{V' \in OPEN}\{g(V') + h(V')\}$
  Remove V from *OPEN*
  if $V = G$
    return the path from T to G via parent pointers
  {Determine how to expand V}
  V.*visitedR* $\leftarrow$ V.*parent*.*visitedR* **or** $V \in \mathcal{R}$
  V.*rworse* $\leftarrow$ V.*parent*.*rworse*
  if $V \notin [BG]^-$ **and** V.*visitedR*
    checkAll $\leftarrow$ **true**
  else
    V.*rworse* $\leftarrow$ V.*rworse* **or** $g(V)+h(V,\mathcal{R})+h(G,\mathcal{R}) >$ *cost*($[TG]^-$)
    if not V.*rworse*
      checkAll $\leftarrow$ **true**
    else if $V \in [BG]^-$
      checkAll $\leftarrow$ **false**
    else
      continue
  for each outgoing edge (V, V')
    if not *checkAll* **and** $V' \neq succ(V, [BG]^-)$
      continue
    if $g(V) + c(V, V') < g(V')$
      $g(V') \leftarrow g(V) + c(V, V')$
      $\pi(V') \leftarrow V$
      Insert or update V' in *OPEN*
return NULL

CPU, an NVIDIA GeForce GTX 680 GPU, and 16 GB of RAM. Only one CPU core was used in Section V-A.

A. DPA* versus A*

We compared the running times of DPA* and A* in the environments shown in Figure 3. Details of the environments and their ECM navigation meshes can be found in Table I. In particular, note that the environments are *not* grid maps.

In each environment, we defined a number of dynamic obstacles (squares of 2×2 m). For each such obstacle O, we performed the following steps:

1) Create 500 pairs of random positions (s, g) that do not intersect the environment or O.
2) For each position pair (s, g), compute a shortest path from s to g in the ECM, using an agent radius of 0.7 m (a size that fits through all passages).
3) Insert O into the ECM dynamically. For each position pair, recompute the path using both DPA* and A*.
4) Delete O dynamically. For each position pair, recompute the path using DPA*. (We can skip regular A*: it would give the same result as in Step 2.)

We always performed all steps for one obstacle before moving on to the next obstacle; hence, the environments contained at most one dynamic obstacle at a time. The average running time of all dynamic insertions in the ECM was 0.34 ms (σ=0.16); the average time for deletions was 1.79 ms (σ=0.89). Table I summarizes the performance of DPA* compared to A* for each re-planning scenario, except 'insertion + path unaffected' for which DPA* immediately terminates. We computed the *improvement* as $(A - D)/A \cdot 100\%$, where A is the sum of all A* times and D is the sum of all DPA* times, over all trials that fit in one scenario. This is a good indication of the time that can be gained by using DPA* instead of A* on a crowd with random characters.

DPA* performs fewer operations on the A* open list in exchange for overhead, e.g. for finding the affected part of a path, and for estimating the distance to $\mathcal{R}$. In the *Military* environment, the ECM graph was too small for this to be benificial, and regular A* was typically slightly faster. In other environments, the improvement remains small except in the 'deletion + path unaffected' scenario, which yielded a 62% improvement in *Zelda4x4*. Hence, DPA* is good at checking whether an unaffected path is still optimal after a deletion. We also observed that the algorithm is particularly fast when the deletion is farther away from the path.

We repeated this experiment with the extra constraint that all start positions lie in the *visibility polygon* [35] of the dynamic obstacle's center of mass. This simulates the effect that agents re-plan when they *see* the event. This greatly improved the results for affected paths, with improvements of 46% (insertions) and 23% (deletions) in *Zelda4x4*. Figure 4 compares the running times of A* and DPA* for each path length in *Zelda4x4*. Naturally, longer paths take more time to compute, but they often allow for more pruning. The paper's supplementary video shows visual examples of visibility-based re-planning compared to instantaneous re-planning.

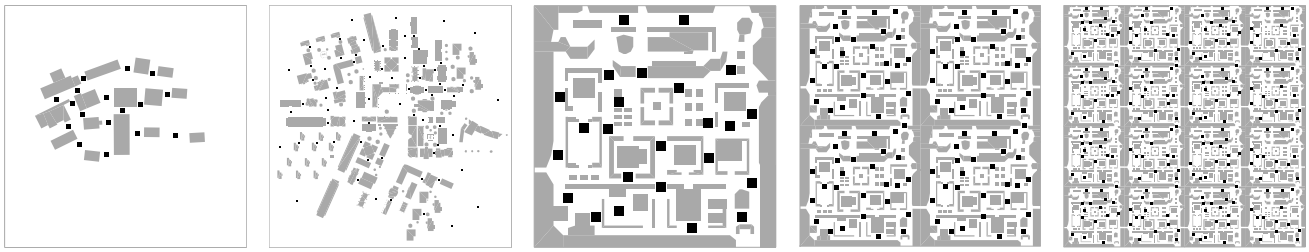
We conclude that DPA* is particularly useful for handling dynamic deletions that did not affect the initial path, and for handling dynamic insertions when the agent can see them. The improvement upon A* is generally better in larger graphs, in which memory-based algorithms are more likely to be unfeasible when simulating a large crowd.

B. Crowd Simulation

Finally, we have integrated DPA* in an ECM-based crowd simulation with smooth path following [36] and collision avoidance between agents [18]. In line with common practice, our simulation uses Euler integration at fixed intervals of $0.1s$. Note that the *visualization* framerate can be higher. The software can simulate over $10{,}000$ agents in real-time using 6 CPU cores in parallel. Obstacles can be added and removed interactively; the crowd responds by re-planning using DPA*. When using visibility as a trigger, re-planning actions are automatically divided over time, allowing real-time performance. For visual examples, we invite the reader to watch this paper's supplementary video.

VI. Conclusions and Future Work

In simulations and games, a dynamic navigation mesh represents an environment in which obstacles are inserted or deleted at runtime. After a dynamic event, agents in a virtual crowd should re-plan their paths. Many re-planning

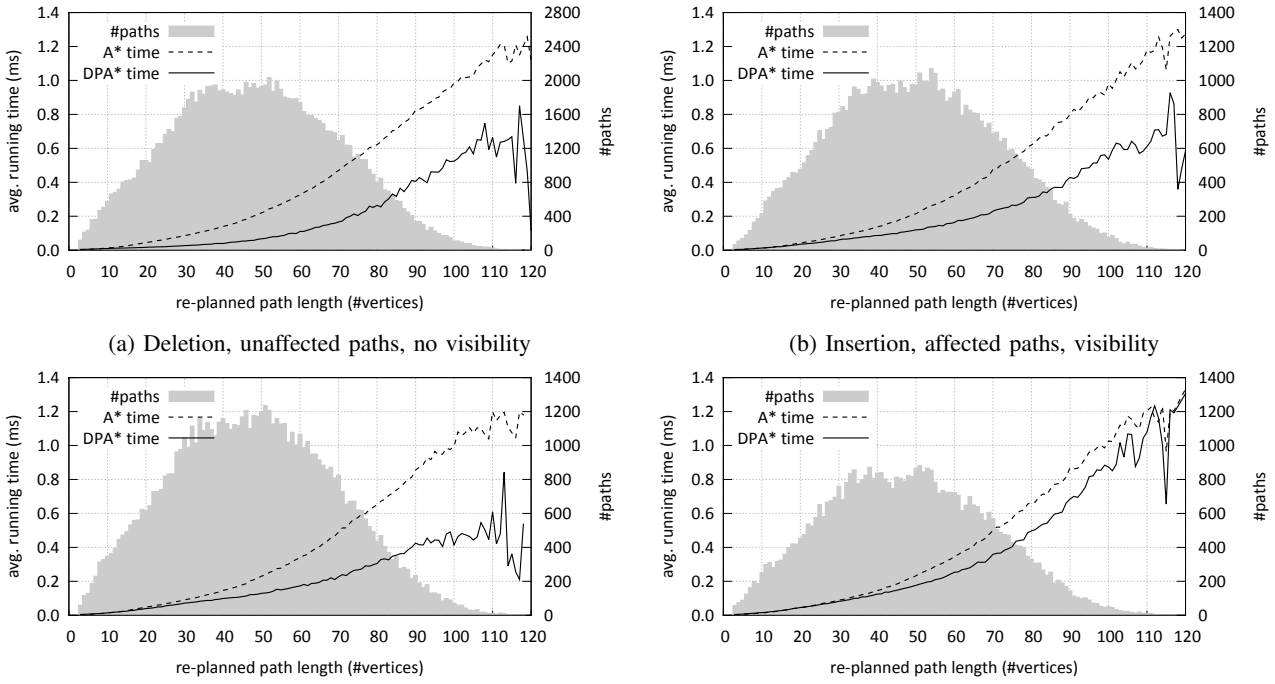


(a) *Military* (b) *City* (c) *Zelda* (d) *Zelda2x2* (e) *Zelda4x4*

Fig. 3: The environments used in our experiment. Static geometry is shown in gray; dynamic obstacles are shown in black.

Environment / ECM					Improvement of DPA* over A*					
					Standard			Source in visibility polygon		
Name	Size (m)	Vertices	Edges	Dynamic obstacles	Insertion, affected	Deletion, unaffected	Deletion, affected	Insertion, affected	Deletion, unaffected	Deletion, affected
Military	200x200	56	70	17	-13.35%	-17.79%	-17.49%	-4.64%	-18.45%	-19.15%
City	500x500	1451	1631	70	-10.17%	40.52%	0.82%	16.87%	34.92%	7.27%
Zelda	100x100	288	343	24	-6.86%	17.67%	-7.57%	11.19%	10.79%	-5.06%
Zelda2x2	200x200	1144	1368	106	-2.39%	41.25%	2.56%	25.46%	22.39%	8.67%
Zelda4x4	400x400	4560	5464	235	2.62%	62.55%	14.84%	46.35%	46.90%	23.41%

TABLE I: Details of the experimental environments. The third and fourth columns show the complexity of the ECM graph without dynamic obstacles. The fifth column shows the number of dynamic obstacles (the black squares in Figure 3). The remaining columns denote the improvement of DPA* over A*, computed as described in Section V-A. A negative percentage (gray) means that A* was faster combined over all trials; a positive percentage (black) means that DPA* was faster.



(a) Deletion, unaffected paths, no visibility

(b) Insertion, affected paths, visibility

(c) Deletion, unaffected paths, visibility

(d) Deletion, affected paths, visibility

Fig. 4: Results of the re-planning experiment in *Zelda4x4*. The horizontal axis denotes the length (the number of vertices) of the re-planned path. The left vertical axis denotes the average running time of A* or DPA* for each path length. The gray histogram and right vertical axis show how often each path length occurred. These figures correspond to the improvements of 62.55%, 46.35%, 46.90%, and 23.41% in Table I.

algorithms are efficient in other applications, but they are not designed for large crowds or structural changes in the search space. In this paper, we have presented *Dynamically Pruned A** (DPA*), which re-plans a path by adding pruning rules to A*, using only the old path and its relation to the dynamic event. This relation is modelled by four possible re-planning *scenarios*; DPA* uses different rules in each scenario. The algorithm is defined for arbitrary graphs and costs, and it yields optimal paths when using admissible heuristics. Its focus is different to that of other re-planning algorithms: DPA* is primarily meant as an improvement of A* for applications that have limited memory per agent.

Experiments show that A* is faster in small graphs, but that DPA* can greatly decrease the re-planning time in complex environments. Our algorithm is particularly efficient for checking whether a path is still optimal after a dynamic deletion, and for responding to a dynamic insertion when it is within the agent's visibility range. In conclusion, DPA* is an intuitive extension of A* that can improve real-time crowd simulation in large dynamic environments.

In the future, we would like to extend DPA* to handle multiple dynamic events without having to re-plan for each event. We also want to simulate incomplete knowledge in the crowd by giving each agent its own set of known and unknown events. Currently, agents know about all events when re-planning because they always use the most recent version of the mesh. Furthermore, we want to explore how this knowledge propagates through the crowd, e.g. how agents can recognize events via the behavior of others. Finally, we are interested in other dynamic geometry, e.g. moving platforms that connect to different areas at different points in time. This asks for new types of navigation meshes and planning algorithms.

REFERENCES

[1] P. Hart, N. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[2] C. Hernández, T. Uras, S. Koenig, J. Baier, X. Sun, and P. Meseguer, "Reusing cost-minimal paths for goal-directed navigation in partially known terrains," *Autonomous Agents and Multi-Agent Systems*, pp. 1–46, 2014.

[3] L. Kavraki, P. Švestka, J.-C. Latombe, and M. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *IEEE Transactions on Robotics and Automation*, vol. 12, no. 4, pp. 566–580, 1996.

[4] S. LaValle, *Planning Algorithms*. Cambridge University Press, 2006.

[5] W. Lee and R. Lawrence, "Fast grid-based path finding for video games," in *Advances in Artificial Intelligence*, ser. Lecture Notes in Computer Science. Springer, 2013, vol. 7884, pp. 100–111.

[6] R. Wein, J. van den Berg, and D. Halperin, "The Visibility-Voronoi complex and its applications," in *Proc. 21st Annual ACM Symposium on Computational Geometry*, 2005, pp. 63–72.

[7] R. Geraerts, "Planning short paths with clearance using Explicit Corridors," in *Proc. IEEE Int. Conf. on Robotics and Automation*, 2010, pp. 1997–2004.

[8] M. Kallmann, "Navigation queries from triangular meshes," in *Proc. 3rd Int. Conf. on Motion in Games*, 2010, pp. 230–241.

[9] J. Pettré, J. Laumond, and D. Thalmann, "A navigation graph for real-time crowd animation on multilayered and uneven terrain," in *Proc. First Int. Workshop on Crowd Simulation*, 2005, pp. 81–89.

[10] W. van Toll, A. Cook IV, and R. Geraerts, "Navigation meshes for realistic multi-layered environments," in *IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, 2011, pp. 3526–3532.

[11] R. Oliva and N. Pelechano, "NEOGEN: Near optimal generator of navigation meshes for 3D multi-layered environments," *Computers & Graphics*, vol. 37, no. 5, pp. 403–412, 2013.

[12] M. Mononen, "Recast Navigation," https://github.com/memononen/recastnavigation/, 2014.

[13] N. Jaklin, A. Cook IV, and R. Geraerts, "Real-time path planning in heterogeneous environments," *Computer Animation and Virtual Worlds*, vol. 24, no. 3, pp. 285–295, 2013.

[14] D. Helbing and P. Molnár, "Social force model for pedestrian dynamics," *Physical Review E*, vol. 51, no. 5, pp. 4282–4286, 1995.

[15] C. Reynolds, "Steering behaviors for autonomous characters," in *Proc. Game Developers Conference*, 1999, pp. 763–782.

[16] I. Karamouzas and M. Overmars, "A velocity-based approach for simulating human collision avoidance," in *Proc. 10th Int. Conf. on Intelligent Virtual Agents*, 2010, pp. 180–186.

[17] J. van den Berg, S. Guy, M. Lin, and D. Manocha, "Reciprocal n-body collision avoidance," in *Proc. 14th Int. Symposium on Robotics Research*, 2011, pp. 3–19.

[18] M. Moussaïd, D. Helbing, and G. Theraulaz, "How simple rules determine pedestrian behavior and crowd disasters," in *Proc. National Academy of Science*, vol. 108, 2011, pp. 6884–6888.

[19] M. Kallmann and M. Matarić, "Motion planning using dynamic roadmaps," in *Proc. IEEE Int. Conf. on Robotics and Automation*, 2004, pp. 4399–4404.

[20] A. Sud, R. Gayle, E. Andersen, S. Guy, M. Lin, and D. Manocha, "Real-time navigation of independent agents using adaptive roadmaps," in *Proc. ACM Symposium on Virtual Reality Software and Technology*, 2007, pp. 99–106.

[21] D. Hale and G. Youngblood, "Dynamic updating of navigation meshes in response to changes in a game world," in *Proc. Int. Florida Artificial Intelligence Research Society Conference*, 2009.

[22] W. van Toll, A. Cook IV, and R. Geraerts, "A navigation mesh for dynamic environments," *Computer Animation and Virtual Worlds*, vol. 23, no. 6, pp. 535–546, 2012.

[23] S. Koenig and M. Likhachev, "D* Lite," in *Proc. AAAI Conf. of Artificial Intelligence*, 2002, pp. 476–483.

[24] S. Koenig, M. Likhachev, and D. Furcy, "Lifelong Planning A*," *Artificial Intelligence*, vol. 155, no. 1-2, pp. 93–146, 2004.

[25] X. Sun and S. Koenig, "The fringe-saving A* algorithm - a feasibility study," in *Proc. Int. Joint Conference on Artificial Intelligence*, 2007, pp. 2391–2397.

[26] S. Aine and M. Likhachev, "Anytime Truncated D*: Anytime replanning with truncation," in *Proc. 6th Int. Symposium on Combinatorial Search*, 2013, pp. 2–10.

[27] M. Likhachev, D. Ferguson, G. Gordon, A. Stentz, and S. Thrun, "Anytime Dynamic A*: An anytime, replanning algorithm," in *Proc. Int. Conf. on Automated Planning and Scheduling*, 2005, pp. 262–271.

[28] J. van den Berg, D. Ferguson, and J. Kuffner, "Anytime path planning and replanning in dynamic environments," in *Proc. IEEE Int. Conf. on Robotics and Automation*, 2006, pp. 2366–2371.

[29] M. Kapadia, A. Beacco, F. Garcia, V. Reddy, N. Pelechano, and N. Badler, "Multi-domain real-time planning in dynamic environments," in *Proc. 12th ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, 2013, pp. 115–124.

[30] M. Phillips, A. Dornbush, S. Chitta, and M. Likhachev, "Anytime incremental planning with E-Graphs," in *Proc. IEEE Int. Conf. on Robotics and Automation*, 2013, pp. 2436–2443.

[31] X. Sun, S. Koenig, and W. Yeoh, "Generalized Adaptive A*," in *Proc. Int. Joint Conference on Autonomous Agents and Multiagent Systems*, 2008, pp. 469–476.

[32] C. Hernández, X. Sun, S. Koenig, and P. Meseguer, "Tree Adaptive A*," in *Proc. Int. Joint Conference on Autonomous Agents and Multiagent Systems*, 2011, pp. 123–130.

[33] C. Hernández, J. Baier, and R. Asin, "Making A* run faster than D*-Lite for path-planning in partially known terrain," in *Proc. Int. Conf. on Automated Planning and Scheduling*, 2014.

[34] F. Preparata, "The medial axis of a simple polygon," in *Mathematical Foundations of Computer Science*. Springer, 1977, vol. 53, pp. 443–450.

[35] S. Ghosh, *Visibility Algorithms in the Plane*. Cambridge University Press, 2007.

[36] I. Karamouzas, R. Geraerts, and M. Overmars, "Indicative routes for path planning and crowd simulation," in *Proc. 4th Int. Conf. on Foundations of Digital Games*, 2009, pp. 113–120.