COMPUTER ANIMATION AND VIRTUAL WORLDS
Comp. Anim. Virtual Worlds 2015; **26**:119–139
Published online 27 October 2014 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cav.1622

SPECIAL ISSUE PAPER

Planning approaches to constraint-aware navigation in dynamic environments

Kai Ninomiya[1], Mubbasir Kapadia[2*†], Alexander Shoulson[1], Francisco Garcia[2] and Norman Badler[1]

[1] Computer and Information Science, University of Pennsylvania, Philadelphia, PA, USA
[2] Computer Science, Rutgers University, New Brunswick, New Jersey

ABSTRACT

Path planning is a fundamental problem in many areas, ranging from robotics and artificial intelligence to computer graphics and animation. Although there is extensive literature for computing optimal, collision-free paths, there is relatively little work that explores the satisfaction of spatial constraints between objects and agents at the global navigation layer. This paper presents a planning framework that satisfies multiple spatial constraints imposed on the path. The type of constraints specified can include staying behind a building, walking along walls, or avoiding the line of sight of patrolling agents. We introduce two hybrid environment representations that balance computational efficiency and search space density to provide a minimal, yet sufficient, discretization of the search graph for constraint-aware navigation. An extended anytime dynamic planner is used to compute constraint-aware paths, while efficiently repairing solutions to account for varying dynamic constraints or an updating world model. We demonstrate the benefits of our method on challenging navigation problems in complex environments for dynamic agents using combinations of hard and soft, attracting and repelling constraints, defined by both static obstacles and moving obstacles. Copyright © 2014 John Wiley & Sons, Ltd.

KEYWORDS
path planning; spatial constraints; navigation; anytime dynamic planning; potential fields

***Correspondence**
Mubbasir Kapadia, Computer and Information Science, University of Pennsylvania, PA, USA.
E-mail: mubbasir.kapadia@gmail.com
†Mubbasir was at University of Pennsylvania while conducting this research.

1. INTRODUCTION

The efficient computation of free movement paths is a fundamental requirement in many disciplines, including robotics, artificial intelligence, computer animation, and games. Robotic agents need to perceive and maintain an internal model of their environment while computing collision-free paths that efficiently navigate them to their destinations. Populating virtual environments with autonomous agents is a key step in making them appear more lifelike and feel more immersive. To contribute to this goal, the autonomous agents themselves must be capable of environment reasoning and pathfinding capabilities, a cognitive ability that serves as the foundation for character animation and behavior synthesis.

There exists a large body of work in path planning research [1], with many proposed solutions that balance path optimality and computational efficiency. These approaches produce trajectories that minimize total distance traveled and use reactive policies to account for constraints (e.g., collision avoidance and inter-agent relationships). This produces locally correct results with no guarantees on constraint satisfaction or global optimality. In addition, it does not scale well to handle constraint combinations and generally can result in non-halting behaviors and inability to solve many problems. Computing global trajectories that account for spatial constraints with respect to obstacles and other agents is still a challenging problem [2] that is of value to the community.

This paper presents a planning approach for constraint-aware navigation that enables autonomous agents to be more aware of the semantics of objects in the environment and thus interpret high-level navigation goals with dynamic and meaningful spatial path constraints. At a personal scale, such constraints may include the following: specific location investigation ('check behind the building'), implemented as a goal state; dynamic agent evasion and stealth ('avoid being seen') and organization ('stay between these two guys'), both implemented using dynamic constraints; or instructions with path requirements for the mission

Copyright © 2014 John Wiley & Sons, Ltd.

('follow the road'), implemented using obstacles or a series of goals. Our approach scales up to larger scenarios as well, allowing constraints such as neighborhood awareness ('avoid this part of the map') or time-of-day awareness ('stay on main roads at night'), implemented using time-dependent constraint weights.

Our initial hybrid discretization of the environment combines the computational benefits of triangulations with a dense uniform grid, ensuring sufficient resolution to account for dynamic constraints. Planning results using this hybrid discretization is highly dependent upon the quality of the environment triangulations. In order to create a more reliable navigation graph, we devise an 'adaptive highway' constraint-dependent planning domain, which has an increased branching factor (and therefore denser environment discretization) in areas affected by constraints.

Hard constraints, which must be satisfied, effectively prune invalid transitions in the search graph (using infinite transition costs), whereas soft constraints (attractors or repellers) have a multiplicative effect on the cost of choosing a transition. Constraints are represented as continuous potential-like fields, which can be easily superimposed to calculate the cumulative effect of multiple constraints in the same region and can be efficiently queried during search exploration. An extended anytime dynamic planner is used to compute constraint-aware paths while efficiently repairing solutions to account for dynamic constraints (e.g., other agents).

This paper makes the following contributions:

- Two alternative environment representations that balance computational efficiency and discretization fineness to provide a minimal-yet-sufficient discretization of the search graph for constraint-aware navigation. (i) A 'highway'-based hybrid graph combines an existing sparse navigation mesh with a runtime-generated dense graph to add resolution where necessary. (ii) An 'adaptive highway' graph, which uses knowledge of the constraint specification in order to improve the accuracy near constraints by adaptively adjusting domain density via branch pruning.
- A method of constraint specification using simple prepositional phrases, which can be easily concatenated to specify complex custom sets of constraints.
- A quantitative description of qualitative motion constraint specifications that can be applied generally to any cost-minimizing spatial pathfinding method.
- A real-time anytime dynamic planning framework that computes trajectories adhering to spatial constraints on both static obstacles and dynamic agents while efficiently repairing plans to accommodate dynamic constraint changes.

To demonstrate the benefits of our method, we present challenging navigation problems in complex environments using combinations of constraints on static obstacles and dynamic agents, including various combinations of hard, soft, attracting, and repelling constraints.

2. RELATED WORK

Depending upon application requirements, a variety of navigation approaches have been proposed [1] for autonomous agents, some of which are described in the succeeding text.

Potential fields. The approach of potential fields [3–7] generates a global field for the entire landscape where the potential gradient is contingent upon the presence of obstacles and distance to goal. These methods suffer from local minima where the agents can get stuck and never reach the goal. Because a change in target or environment requires significant re-computation, these navigation methods are generally confined to systems with non-changing goals and static environments. Dynamic potential fields [8] have been used to integrate global navigation with moving obstacles and people, efficiently solving the motion of large crowds without the need for explicit collision avoidance. The work of Kapadia *et al.* [9,10] uses local variable-resolution fields to mitigate the need for computing uniform global fields for the whole environment and uses best-first search techniques to avoid local minima.

Continuous weighted-region navigation. Various continuous navigation methods in non-uniformly weighted space have been developed. One continuous Dijkstra technique finds the shortest paths using the principles of Snell's law [11]. Methods have been shown for finding exact solutions to simpler weighted-region problems [12]. Suboptimal solutions can be found using an efficient approximation Algorithm [13].

Discrete graph-based search. Discrete search methods such as A* [14–16] are robust and simple to implement, with strict guarantees on optimality and completeness of solution. Hence, they represent a popular and widely used method for path planning in commercial systems such as games. However, the performance and quality of the obtained paths greatly depend on the resolution of the discretization, with coarse resolutions producing low-quality paths and fine resolution grids proving to be computationally prohibitive for real-time applications. Many triangulation-based navigation meshes have been developed for portability and applicability [17] to allow navigation of agents with arbitrary clearance [18], to improve performance using suboptimal meshes [19,20], to enable crowd animations in uneven terrain [21], and to permit real-time dynamic modification of the mesh [22]. In addition, graphs can be generated or modified for optimized suboptimal searches, for example, using highway node routing [23] and contraction hierarchies [24].

Other more advanced A*-based search algorithms have been developed. The D* Lite [25] graph search algorithm is able to efficiently repair computed paths to accommodate dynamic changes in the environment. ARA* [26] provides

Comp. Anim. Virtual Worlds 2015; **26**:119–139 © 2014 John Wiley & Sons, Ltd.
DOI: 10.1002/cav

anytime solution guarantees with strict bounds on suboptimality. These two methods provide the basis for AD* [27], which combines the properties of D* Lite and ARA*, to provide an efficient real-time search technique that is applicable in dynamic environments.

Local collision avoidance. There is an extensive work [28] that relies on local goal-directed collision avoidance for simulating large crowds. This includes rule-based approaches [29], social forces [30,31], predictive methods [32–34], local fields [9], and planning-based approaches [35,36]. The work of Schuerman *et al.* [37] externalizes steering logic to enforce local constraints for group formations.

Navigation with constraints. The work of Xu and Badler [38] describes a list of representative prepositions for constraining motion trajectories in goal-directed navigation. The work of André [39] analyzes the semantics of spatial relations, including 'along' and 'past', to characterize the path of moving objects. In addition, several search methods based on homotopy classes have been proposed. Bhattacharya *et al.* [40,41] explore the use of homotopy classes of trajectories in graph-based search for path planning with constraints. This method is extended by Bhattacharya *et al.* [40] to handle 3D spaces. A homotopy class-based approach to A*, called *HA** [42], ensures optimality and directs the search by exploring areas that satisfy a given homotopy class. The work of Phillips *et al.* [43] demonstrates constrained manipulation using experience graphs. The works of Geraerts [44] and Kallman [18] embed additional information in the underlying environment representation to efficiently compute the shortest paths with clearance constraints. Sturtevant [45] explores the integration of human relationships into path planning in games, incorporating 'relationship distance' and line of sight as positive and negative weights in the path planning problem.

Comparison to prior work. Our method provides a generic way to specify spatial constraints, including constraints on dynamic objects (such as other agents) modeled as local artificial potential fields, which contribute to the cost of a node in the search graph. In its implementation, our constraint-handling method (weighted planning space) is similar to that of Sturtevant [45] but more generalized. We develop two environment representations: a 'hybrid' domain, which combines the benefits of triangulations and a uniform grid and an 'adaptive highway' domain, which controls graph expansion using its knowledge of the constraint system. In comparison with Xu and Badler [38], we utilize an anytime dynamic planner that can efficiently repair solutions to accommodate constraint changes. The resulting trajectory does not depend on the shape of an object but rather on the location and affecting area of the constraints. Thus, we do not encounter the issues described in the work of André *et al.* [39].

Extension. This invited journal submission extends our earlier MIG2013 conference publication [46] as follows. We introduce the new advanced 'adaptive highway' planning domain that integrates knowledge of constraint specifications to improve computational performance and plan quality. This includes performance results and comparison with other domains. We have revamped our formulation of the quantitative representation of our constraint specifications, replacing the original multiplier field formulas with a more cleanly expressed definition that eliminates several extraneous constant definitions. Finally, we have applied our work to more accurate human relationship constraints (namely, personal distance and line of sight) and to large-scale and spatiotemporal planning problems (e.g., map navigation and time-of-day-dependent constraints).

3. PROBLEM DEFINITION

The problem domain $\Sigma = \langle \mathbf{S}, \mathbf{A} \rangle$ defines the set of all possible states **S** and the set of permissible transitions **A**. Every problem instance **P**, for a particular domain Σ, is defined as $\mathbf{P} = \langle \Sigma, s_{\text{start}}, s_{\text{goal}}, \mathbf{C} \rangle$, where $(s_{\text{start}}, s_{\text{goal}})$ are the start and goal state, and **C** is the set of active (hard and soft) constraints. A hard constraint is used to prune transitions in **A**. For example, consider a flower bed that *must* not be stepped upon. A hard constraint could be specified for that area, pruning every transition that could violate this restriction. A soft constraint influences the costs of actions in the action space and can tend the agent toward or away from a certain region in space. A planner generates a plan, $\Pi\left(s_{\text{start}}, s_{\text{goal}}\right)$, which is a sequence of states from s_{start} to s_{goal} that adheres to **C**.

4. ENVIRONMENT REPRESENTATION

In this section, we describe the discretized environment representation that we use for constraint-aware pathfinding. The challenge we encounter is as follows. A coarse-resolution representation, such as a mesh triangulation-based approach, is often used to facilitate efficient search. However, it cannot accommodate all constraints as it has insufficient resolution in regions of the environment where constraints may be specified. Because dynamic constraints are not known ahead of time, it is impossible to simply increase triangulation density near constraints. A sufficiently dense uniformly distributed graph representation of the environment can account for all constraints (including dynamic objects) but is not efficient for large environments.

To avoid these limitations, we explore two alternative methods of generating environment representations. First, we experimented with a 'hybrid' search graph that has sufficient resolution while still accelerating search computations by exploiting longer, coarser transitions to improve suboptimal planning speed in certain situations. More efficient results were seen with an 'adaptive highway' domain, operating in the same *state* space as the dense environment representation but with dynamically reduced branching at states unaffected by constraint effects. This provides bounded suboptimality: the reduced-branching graph is,

at worse, no more coarser than a lower resolution dense uniform graph.

4.1. Triangulation

We define a simple triangulated representation of free space in the environment represented by $\Sigma_{\text{tri}} = \langle \mathbf{S}_{\text{tri}}, \mathbf{A}_{\text{tri}} \rangle$, where elements of $\mathbf{S}_{\text{tri}}$ are the midpoints of the edges in the navigation mesh and elements of $\mathbf{A}_{\text{tri}}$ are the six directed transitions per triangle (two bi-directional edges for each pair of states). This simple triangulation can be easily replaced by other more advanced navigation meshes [17]–[21] and produces a low-density representation of the state and action space. Figure 1a illustrates Σ_{tri} for a simple environment. The triangulation domain Σ_{tri} provides a coarse-resolution discretization of free space in the environment, facilitating efficient pathfinding. However, the resulting graph is too sparse to represent paths adhering to constraints such as spatial relations to an object.

To offset this limitation, we can annotate objects in the environment with additional geometry to describe relative spatial relationships (e.g., `Near`, `Left`, and `Between`) We use these annotations to generate additional triangles in the mesh, which expand Σ_{tri} to include states and transitions that can represent these spatial relations. Annotations, and the corresponding triangulation, are illustrated in Figure 1(b). These annotations are useful for constraints relative to static objects; however, Σ_{tri} cannot account for dynamic objects as the triangulation cannot be efficiently recomputed on the fly. To handle dynamic constraints, we utilize a dense, uniform graph representation described in the succeeding text.

4.2. Dense Uniform Graph

To generate $\Sigma_{\text{dense}} = \langle \mathbf{S}_{\text{dense}}, \mathbf{A}_{\text{dense}} \rangle$, we densely sample points in the 3D environment, separated by a uniform distance d_{grid}, which represents the graph discretization. For each of these points, we add a state to $\mathbf{S}_{\text{dense}}$ if it is close to the navigation mesh (within $\frac{\sqrt{3}}{2} d_{\text{grid}}$ of the nearest point) and clamp it to that point (Figure 1(c)). This allows the method to work well in environments with slopes, stairs, and multiple levels. Although all of our environments are locally 2D, the method could be used directly

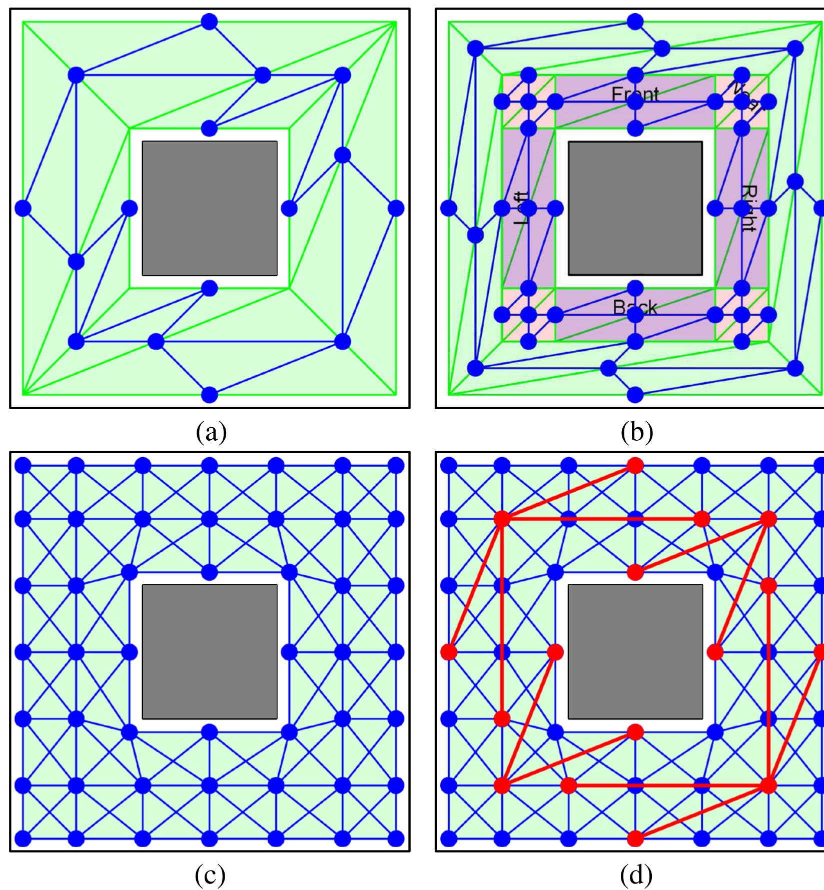


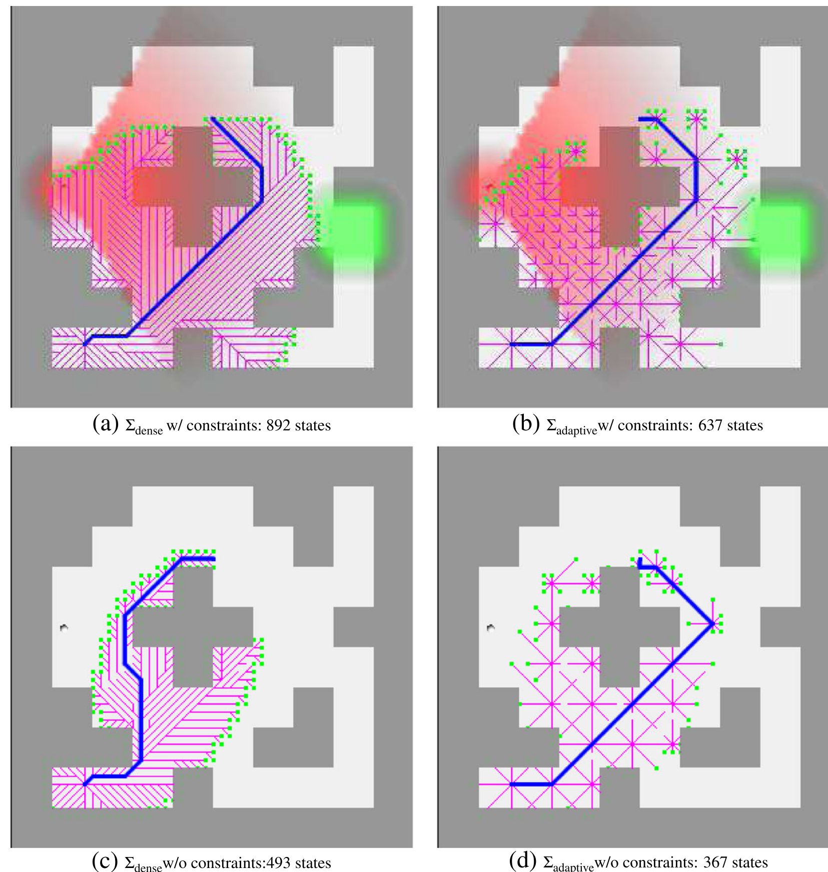
Figure 1. (a) Environment triangulation Σ_{tri}. (b) Object annotations, with additional nodes added to Σ_{tri}, to accommodate static spatial constraints. (c) Dense uniform graph Σ_{dense} for same environment. (d) A hybrid graph Σ_{hybrid} of (a) Σ_{tri} and (c) Σ_{dense}; highways (newly inserted transitions) are indicated in red.

for 3D domains by testing against a volume rather than a mesh surface. Examples are shown in a completely flat 2D for illustrative purposes. Because the graph is sampled in 3D, each state in $\mathbf{S}_{\text{dense}}$ could have a maximum of 26 neighbors; however, in practice, each state has no more than eight neighbors if the domain operates in a locally 2D environment (such as navigation domains for humanoids or ground vehicles). The dense domain Σ_{dense} can be precomputed or generated on the fly, depending on environment size and application requirements. Regardless of how it is implemented, however, a dense domain greatly increases the computational burden of the search due to the increased number of nodes and transitions compared with a sparse domain (Figure 1(a/c)). Expansion examples can be seen in Figure 2(a/c).

4.3. Hybrid Graph

In the first two attempts to mitigate the performance problem of Σ_{dense}, we combine Σ_{dense} and Σ_{tri} to generate a hybrid domain $\Sigma_{\text{hybrid}} = \langle \mathbf{S}_{\text{hybrid}} = \mathbf{S}_{\text{dense}}, \mathbf{A}_{\text{hybrid}} \approx \mathbf{A}_{\text{dense}} \cup \mathbf{A}_{\text{tri}} \rangle$. First, we add all the states and transitions in Σ_{dense} to Σ_{hybrid}. For each state in $\mathbf{S}_{\text{tri}}$, we find the closest state in $\mathbf{S}_{\text{dense}}$, creating a mapping between the state sets, $\lambda : \mathbf{S}_{\text{tri}} \rightarrow \mathbf{S}_{\text{dense}}$. Then, for each transition $(s, s') \in \mathbf{A}_{\text{tri}}$, we insert the corresponding transition $(\lambda(s), \lambda(s'))$ in $\mathbf{A}_{\text{dense}}$ (if it does not already exist). The resulting hybrid domain Σ_{hybrid} has the same states as Σ_{dense} with additional transitions (Figure 1(d)). These transitions are generally much longer than those in $\mathbf{A}_{\text{dense}}$, creating a low-density network of *highways* through the dense graph. This approach is similar in concept to those of highway node routing [23] and contraction hierarchies [24], but we exploit information provided by the navigation mesh.

In Σ_{hybrid}, a pathfinding search can automatically choose highways for long distances and use the dense graph more heavily when the planner has additional time to compute an exact plan. As before, the dense graph allows the planner to find paths that adhere to constraints.



(a) Σ_{dense} w/ constraints: 892 states
(b) Σ_{adaptive} w/ constraints: 637 states
(c) Σ_{dense} w/o constraints: 493 states
(d) Σ_{adaptive} w/o constraints: 367 states

Figure 2. Graph expansion comparison between dense uniform and adaptive highway domains. (a) Dense uniform navigation domain with constraints. (b) Adaptive highway navigation domain with constraints. (c) Dense uniform navigation domain, no constraints. (d) Adaptive highway navigation domain, no constraints. Note in figures (c) and (d) that the resulting path is slightly different. This is because that the adaptive highway domain represents the environment with slightly less accuracy. (a) Σ_{dense} with constraints: 892 states; (b) Σ_{adaptive} with constraints: 637 states; (c) Σ_{dense} without constraints: 493 states; (d) Σ_{adaptive} without constraints: 367 states.

When there is no strong influence of nearby constraints, the planner can take highways to improve its performance. In addition, with a planner such as AD* [27], we can inflate the influence of the heuristic to very quickly produce suboptimal paths that favor highway selection then iteratively improve the path quality by using dense transitions. This method makes it possible to maintain interactive frame rates. The performance benefits of Σ_{hybrid} are described in Section 7.1.

4.4. Adaptive Highway Graph

For practical reasons, we wish to avoid depending heavily on both the quality of the triangulations in Σ_{tri} and the behavior of the anytime dynamic planner for suboptimal performance improvements. To this end, we introduce a uniform search domain with constraint-adapting highways. This graph maintains a high density of search nodes near constraints while automatically pruning transitions when additional accuracy is not needed. The results of this pruning approach are shown and described in Figure 3.

The basis of this domain is a lazily evaluated form of Σ_{dense}. As an example, in a 2D navigation problem, a state in Σ_{dense} has eight neighbors and, therefore, eight potential transitions. However, because high graph resolution is not necessary in obstacle-free, constraint-free areas, it is possible to reduce the graph resolution in many parts of an environment. That is, the resolution of the graph should be dependent upon the absolute amount of constraint weight in that area (the *absolute value weight field*).

In order to create this adaptive change in resolution, we emulate the 'highway' transitions found in Σ_{hybrid}; however, instead of adding long distance transitions, we can take another approach. Because Σ_{adaptive} is aware of the expansion of the planner, it instead 'skips' through nodes in a single direction. That is, from a given state (e.g., the start state), the domain will expand in eight directions, but in the absence of constraints, each of those expanded states may have only one possible transition, which continues in the same direction. The 'skipping' expansion is controlled as follows.

- When a graph state is expanded, it stores a 'remaining number of skips' value into each expanded state. This keeps track of how many skips are left in a given

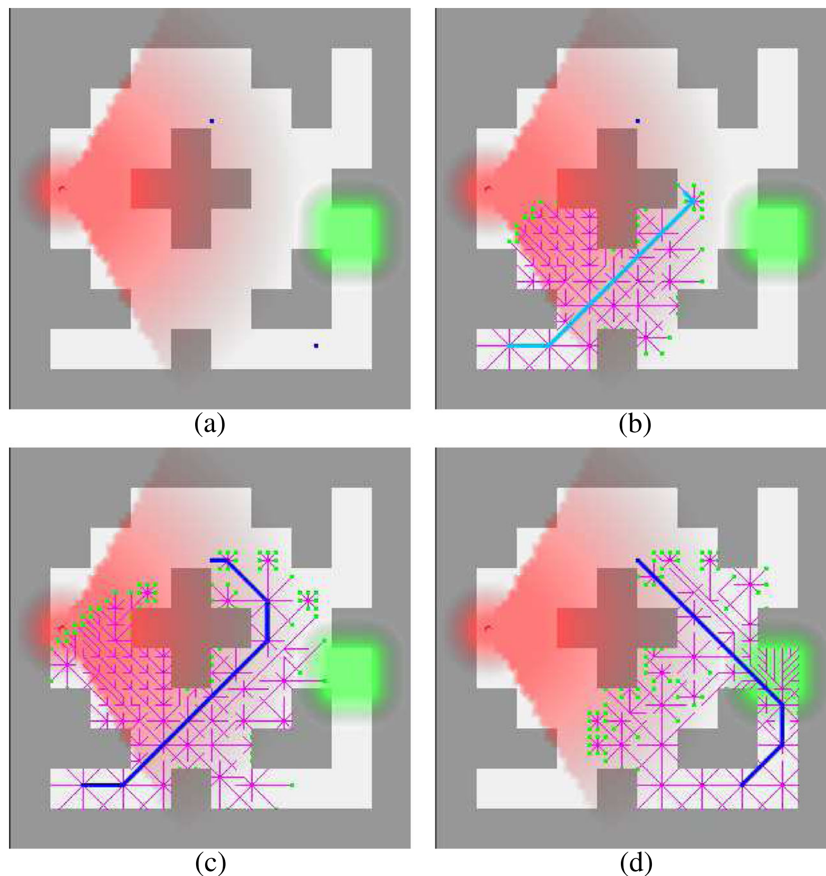


Figure 3. (a) Multiplier field incorporating a negative human relationship constraint [45] and a positive area constraint. (b) 'Partial' path planning result before planner has completed execution. (c) and (d) Example plans showing the 'adaptive highway' planning domain in this particular planning problem.

Comp. Anim. Virtual Worlds 2015; **26**:119–139 © 2014 John Wiley & Sons, Ltd.
DOI: 10.1002/cav

highway, and we limit skipping expansion to (in our examples) at most four transitions in one direction.
- In addition, as the expansion continues skipping in one direction, the cumulative integral of the absolute-value weight field is also tracked. When too much absolute-value weight is encountered, we cancel skipping expansion early to ensure high graph density near constraints.

To ensure correct behavior when states are invalidated, Σ_{adaptive} simply re-expands invalidated areas using the newly computed weight field, discarding the old expansion. The traits of the adaptive highway domain and performance trade-offs are described in Section 7.2.

Observed limitations of the current Σ_{adaptive} might be the subject of future research and are as follows. First, although the branching factor is correctly reduced in unweighted areas, the increased state density persists after the planner's search passes beyond a constraint. This is seen just past the positive constraint in Figure 3(c). A potential fix would be to explicitly reduce the state density in this case. Second, the branch density is dependent upon the amount of absolute constraint weight in an area. Because of this, inside a very large, constant constraint, the graph density will still be high; better results might be achieved by adjusting branching factor based on the rate of change of the weight field. In addition, in the absence of constraints, navigation near edges of obstacles is inaccurate because of the low state density. It may be possible to remedy this by decreasing skip lengths near obstacles and walls. Finally, in Figures 2(c/d), note that the resulting paths are very different. This is because the adaptive highway domain represents the environment with less accuracy in favor of performance improvements, sometimes giving suboptimal results. However, because the skip count in the adaptive highway domain is bounded, this provides a bound on the suboptimality of this method: it will never be less optimal than a domain that always takes the maximum number of skips (i.e., a domain with lower density).

5. CONSTRAINTS

Constraints imposed on how an agent navigates to its destination greatly influence the motion trajectories that are produced and often result in global changes to the paths that cannot be met using existing local solutions. For example, if there is an agent who wishes to stay behind a building or outside another agent's line of sight, our method may choose very circuitous paths in order to satisfy these constraints. Our framework supports hard constraints (obstacles), which must always be met; attractors, which reduce nearby transition costs; and repellers, which increase nearby transition costs.

Problem specification. A constraint-aware planning problem is represented by a start state s_{start}, a goal state s_{goal}, and a set of constraints $\mathbf{C} = \{c_i\}$. Each constraint is defined as $c_i = ((\texttt{In}|\texttt{Near})\ \texttt{Annotation}_i$ with weight $w_i)$, where the weight w_i can be either positive (attracting) or negative (repelling). Hence, a specification can be written as a simple regular language

$$\text{Move from } s_{\text{start}} \text{ to } s_{\text{goal}}\ ((\texttt{In}|\texttt{Near})\ \texttt{Annotation}_i \text{ with weight } w_i) *$$

The terms used in this expression are defined in the next subsection. Despite the simplicity of such a definition, it is important to note its flexibility: both goals and constraints can encode very rich semantics for a wide variety of planning problems. In addition, multiple problem specifications can be chained together to create more complex commands, for example, 'move to the street corner, then patrol the alleyway', where 'patrol' can be described as a repeating series of commands going back and forth between two points.

5.1. Constraint Definitions

Annotations. An annotation is simply a volume of space that allows the user to define the area of influence of a constraint. These volumes can be defined in real-time and may be dynamically calculated from environment and agent information. By attaching annotations to an object in the environment, a user can provide useful positional information. These annotations are used to semantically construct customized prepositional constraints (for example, 'in the grass', 'near the wall', or 'west of the train tracks'). Figure 1(b) illustrates some common annotations: `Back`, `Front`, `Left`, and `Right` for a static object in the environment. The relationships between multiple objects can be similarly described by introducing annotations such as `Between` shown in Figure 4(c).

Dynamic objects can also easily be given spatial annotations. For example, we can denote an agent's `LineOfSight`; a simple approximation of this is shown in Figure 3. Note that it is conceptually simple (and supported by our framework) to calculate 'true' line of sight or other complex annotations, dynamically, as cost calculations can happen during path planner execution. In a world model that supports ray casting, true line of sight can be directly calculated in this way.

Annotations are defined independently from constraints and are used in the definition of a constraint to delimit its area of influence based on objects, agents, and the environment. As such, annotations may be created ahead of time and freely reused by different constraint systems, which may run independently, allowing for rapid authoring of new constraint systems for new situations.

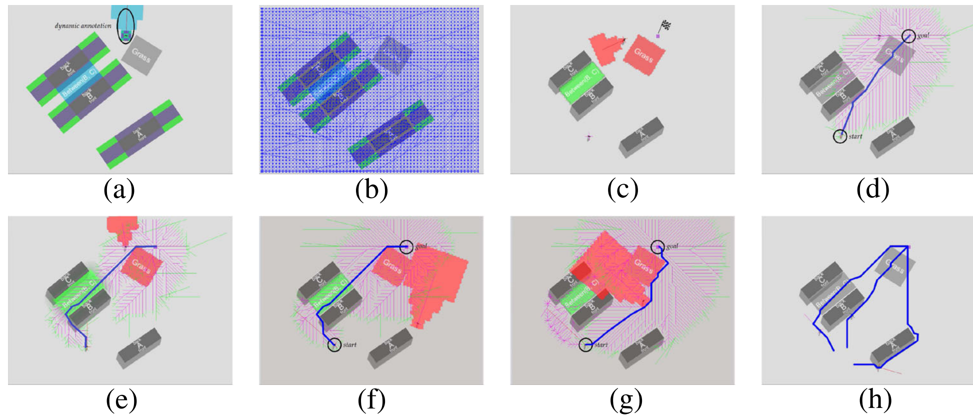


Figure 4. (a) An environment with annotations such as `Front`, `Left`, `Between`, and a simple `LineOfSight`. (b) Transitions in hybrid domain Σ_{hybrid}. (c) A specific problem instance with the following constraints: `Not In` Grass $\wedge$ `Not Near LineOfSight` (Agent) $\wedge$ `Near Between` (B, C). (d) Static optimal path, in absence of constraints. (e) Resulting path produced for problem instance (c). (f).(g) Plan repair to accommodate moving `LineOfSight` constraint. The `Between` constraint is invalidated due to the `LineOfSight` constraint, of higher priority. (h) Multiple characters simultaneously navigate under different constraint specifications, producing different paths from the same start/goal configuration.

Hard constraints. A hard constraint comprises just one field: an annotation. This annotation represents an area in which states in Σ are pruned. Hard constraints can only be `Not` constraints. In order to specify hard attracting constraints, we use a sequence of goals that the agent must navigate to (e.g., go to the mailbox and then come back to the door). Hard constraints prevent all included transitions from being expanded during the search (by giving these transitions an infinite weight), thus producing a plan that always avoids the region. In addition, hard constraints can be used to model dynamic obstacles (any annotation that completely blocks off its contained states).

Soft constraints. A soft constraint specification consists of three fields: (i) a preposition; (ii) an annotation; and (iii) the constraint weight. As soft constraints are the general case encompassing hard constraints (which are `Not In` with $w = -\infty$), they are simply referred to as 'constraints' henceforth.

Preposition. We define two simple prepositions, `Near` and `In`, which define the boundaries of the region of influence. For example, we might wish to navigate `Near` a building (a fuzzily defined area of effect) while making sure that we are not `In` the grass (a well-defined area of effect). These two prepositions gain significant power by leveraging annotations placed in meaningful parts of the environment.

Weight. The weight defines the influence of a constraint and can be positive or negative. For example, one constraint may be a weak preference ($w = 1$), whereas another may be a very strong aversion ($w = -5$); a negative weight indicates a repelling factor. Weights allow us to define the influence of constraints relative to one another (where one constraint may outweigh another), facilitating the superposition of multiple constraints in the same area with consistent results.

5.2. Multiplier Field

Constraints must modify the costs of transitions in the search graph in order to have an effect on the resulting path generated. To achieve this, it is important to maintain several properties:

(1) *The modified cost of a transition must never be negative to ensure that the search technique will be complete.* In our system, the cost of a transition will always be greater than or equal to its unmodified distance cost, even under the influence of attractor constraints. With A* and variants such as AD*, this guarantees optimality (and can prevent infinite loops and other instabilities) when the unmodified distance cost is used as a heuristic cost estimate.
(2) *We must be able to efficiently compute the cost of a transition, influenced by several constraints.* The weighted influence of all constraints at a particular state is simply summed together and added onto the base weight, multiplying the cost.
(3) *Constraints should only affect a limited region of influence.* In our system, soft constraints may have a smooth gradient or a hard edge. We model constraints as artificial potential fields and define the influence of a constraint at a particular position as a function of its distance from the nearest point in the annotation volume. A linear falloff ensures that a constraint has influence only at short distances (A hard constraint only affects states that are within its annotation volume).
(4) *The total cost along a path must be independent of the number of states along that path.* To maintain this property, the cost calculation must be continuous and modeled as a path integral (see following

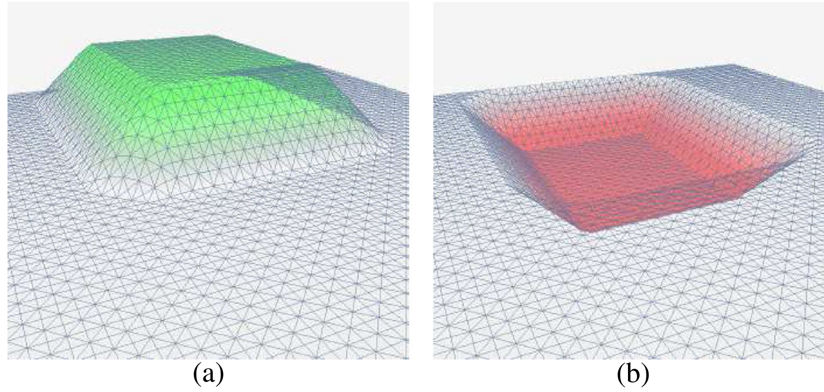


Figure 5. 3D depictions of the additive weight fields $W_c(\vec{x})$ for (a) attracting/positive and (b) repelling/negative constraints. These demonstrate the falloff properties of soft `Near` constraints and provide an intuitive picture of how constraints are quantified.

text). The path integral will always have the same value regardless of path subdivision.

Formulation. The influence of a constraint is defined using a continuous multiplier field $m(\vec{x})$, where $m(\vec{x})$ denotes the multiplicative effect of the constraint at a particular position $\vec{x}$ in the environment. It is important to note that, because of its continuous nature, the multiplier field can be easily translated to any pathfinding system; it is not specific to graph search representations of pathfinding problems. A previous version of this formulation is seen in our original publication [46]. The reworked formulation is thought to be superior because of its straightforwardness and more cleanly expressed definition.

For a single constraint c, the cost multiplier field $m_c(\vec{x})$ is defined as follows:

$$m_c(\vec{x}) = 1.1^{-W_c(\vec{x})}$$

where $W_c(\vec{x})$ is the constraint weight field and constant 1.1 is chosen for convenience[†]. The constraint weight field is defined as a position-dependent weight value for a constraint. For `In` constraints, it has a discrete definition

$$W_c(\vec{x}) = \begin{cases} w : \vec{x} \in \text{annotation}_c \\ 0 : \text{otherwise} \end{cases}$$

whereas for `Near` constraints, it provides a soft falloff with a fixed radius of $|w|$ outside of the annotation

$$W_c(\vec{x}) = w \cdot \max\left(0, \frac{|w| - r_c(\vec{x})}{|w|}\right)$$

where $r_c(\vec{x})$ is the distance between the position $\vec{x}$ and the nearest point in the volume of the annotation on constraint c. Outside of the fixed radius $|w|$, a `Near` constraint has no effect.

[†]It could be any value greater than 1; its value only affects the range of useful w values.

This is especially important for dynamic constraints as we must monitor all the states whose costs are updated while performing plan repair. Explicitly defining the boundary of a constraint limits the number of states that a planner must consider for repair. Multiplier fields for `Near` attractor and repeller are visualized in Figure 5.

Multiple constraints. For a set of constraints $\mathbf{C}$, we define the aggregate cost multiplier field

$$m_{\mathbf{C}}(\vec{x}) = \max\left(1, m_0 \prod_{c \in \mathbf{C}} m_c(\vec{x})\right) = \max\left(1, 1.1^{W_0 - \sum_{c \in \mathbf{C}} W_c(\vec{x})}\right)$$

To accommodate attractor constraints, which reduce cost, we define a 'base' multiplier m_0 or base weight W_0, which is automatically calculated based on the weight values of the constraints in $\mathbf{C}$. This multiplier affects costs even in the absence of constraints, which allows attractors to reduce the cost of a transition while remaining the aforementioned original (Euclidean distance) cost. The resulting cost multiplier is thus limited to be ≥ 1, preserving optimality guarantees of the planner.

Cost multiplier for a transition. The cost multiplier for a transition $(s \rightarrow s')$, given a set of constraints $\mathbf{C}$, is defined as follows:

$$M_{\mathbf{C}}(s, s') = \int_{s \rightarrow s'} m\mathbf{C}(\vec{x})\, d\vec{x}$$

We choose to define this as a path integral because it is generalized to any path, not just a single discrete transition, and because it perfectly preserves cost under any path subdivision. For our graph representation, we estimate the path integral using a four-part Riemann approximation by taking the value of the multiplier field at several points along the transition.

6. PLANNING ALGORITHM

We use anytime dynamic A* [27] as our underlying planner, which combines the incremental planning properties of

D* Lite [25] and the anytime planning properties of ARA* [26] to efficiently repair solutions after world changes and agent movement. It quickly generates an initial sub-optimal plan bounded by an initial inflation factor ϵ_0, which focuses search efforts toward the goal. This initial plan is then improved by lowering the weight of ϵ gradually (while planning) until ϵ reaches 1.0, guaranteeing optimality of the final solution. AD* allows an anytime, dynamic approach to this weighted-region problem (previously solved using various other methods [11–13]). The suboptimal solutions provided by the AD* inflation methods are used to find approximate solutions in real-time.

AD* can interleave planning with execution by allowing the agent to move along the path, handling updates to the start position by performing a backwards search (The planner, however, cannot handle dynamic changes in goal, so in those circumstance, we can simply reset ϵ to its default value and plan from scratch.). Dynamic state changes are efficiently handled by keeping track of states whose costs are inconsistent, re-expanding them to repair the solution. This avoids having to re-plan from scratch every time there is a dynamic event in the environment. For more details on AD*, we refer the readers to the work of Likhachev *et al.* [27] and describe our changes to accommodate constraint satisfaction. Appendix 1 provides the algorithmic details of AD* for reference.

Cost computation. The modified cost of reaching a state s from s_{start}, under the influence of constraints, is computed as follows:

$$g(s_{\text{start}}, s) = g(s_{\text{start}}, s') + \boldsymbol{M}_{\mathbf{C}}\left(s, s'\right) \cdot c(s, s')$$

where $c(s, s')$ is the cost of a transition from $s \rightarrow s'$, and $\boldsymbol{M}_{\mathbf{C}}\left(s, s'\right)$ is the aggregate influence of all constraint multiplier fields, as described in the Section 5.2. This is recursively expanded to produce

$$g(s_{\text{start}}, s) = \sum_{(s_i, s_j) \in \Pi(s_{\text{start}}, s)} \boldsymbol{M}_{\mathbf{C}}\left(s_i, s_j\right) \cdot c(s_i, s_j)$$

which utilizes the constraint-aware multiplier field to compute the modified least cost path from s_{start} to s, under the influence of active constraints $\mathbf{C}$. Each state can keep track of its set of influencing constraints to mitigate the need to exhaustively evaluate every constraint repeatedly. When the area of influence of a constraint changes, the states are efficiently updated, as described in the following text.

Accommodating dynamic constraints. Over time, objects associated with a constraint may change in location, affecting the constraint multiplier field, influencing the search. For example, an agent constrained by a `LineOfSight` constraint may change position, requiring the planner to update the plan to ensure that the constraint is satisfied. Each constraint multiplier field $\vec{x}$ has a region of influence **region**$(m_c, \vec{x})$, which defines the finite set of states $\mathbf{S}_c$ that is currently under its influence.

We take the following approach for moving annotation-based constraints. When a constraint c moves from $\vec{x}_{\text{prev}}$ to $\vec{x}_{\text{next}}$, the union of the states that were previously and currently under its region of influence ($\mathbf{S}_c^{\text{prev}} \cup \mathbf{S}_c^{\text{next}}$) is marked as inconsistent (their costs have changed) and they must be updated. In addition, for states $s \in \mathbf{S}_c^{\text{next}}$, if c is a hard constraint, its cost is $g(s) = \infty$. Algorithm 1 provides the pseudocode for **ConstraintChangeUpdate.** Note that, in the more general case (with non-annotation-based constraints such as 'real' line of sight), the inconsistent region can be detected by changes in weights at every node that may have changed (or other implementation-specific methods).

Finally, if the navigation graph has changed, the routine **UpdateState**(s) is used to recompute the costs of states; see Appendix 1, which is modified slightly from its original definition [27] to incorporate the multiplier fields during cost calculation. Another possible approach would be to periodically update the plan instead of detecting when it is necessary. Our approach has an advantage in more dynamic environments, where periodic re-planning may be too infrequent. In our vehicle example (Figure 6), less frequent updates would result in collisions with the vehicle hazards.

In actual execution, we note that re-planning time can sometimes be considerable. To maintain real-time performance without continuing along the old, incorrect trajectory, the system will report no known solution until a partial or full plan is known (depending on the exact algorithm conventions). As a result, a 'stopping'/'waiting' behavior is observed: an agent may pause its motion while recalculating to find the best way to continue. In most cases, this stopping time will be short. With algorithms able to provide suboptimal paths (such as AD*), the agent may follow these solutions while recalculating.

Algorithm 1 ConstraintChangeUpdate$(c, \vec{x}_{\text{prev}}, \vec{x}_{\text{next}})$

1: $\mathbf{S}_c^{\text{prev}} =$ **region**$(m_c, \vec{x}_{\text{prev}})$
2: $\mathbf{S}_c^{\text{next}} =$ **region**$(m_c, \vec{x}_{\text{next}})$
3: **for all** $s \in \mathbf{S}_c^{\text{prev}} \cup \mathbf{S}_c^{\text{next}}$ **do**
4:     **if** `pred`$(s) \bigcap$ `VISITED` $\neq$ `NULL` **then**
5:         **UpdateState**s
6:     **if** $s' \in \mathbf{S}_c^{\text{next}} \wedge c \in \mathbf{C}_h$ **then** $g(s') = \infty$
7:         **if** $s' \in$ `CLOSED` **then**
8:             **for all** $s'' \in$ `succ`(s') **do**
9:                 **if** $s'' \in$ `VISITED` **then**
10:                     **UpdateState**s''

7. RESULTS

Our initial code base is implemented in C# and uses the Unity game engine. The ADAPT platform [47] was used for character animation where applicable. Our revised code base is implemented as a stand-alone .NET library in C# along with a layer for integrating with Unity. This revised code includes implementations of the various planners (A*, ARA*, AD*, etc.) and the system for handling

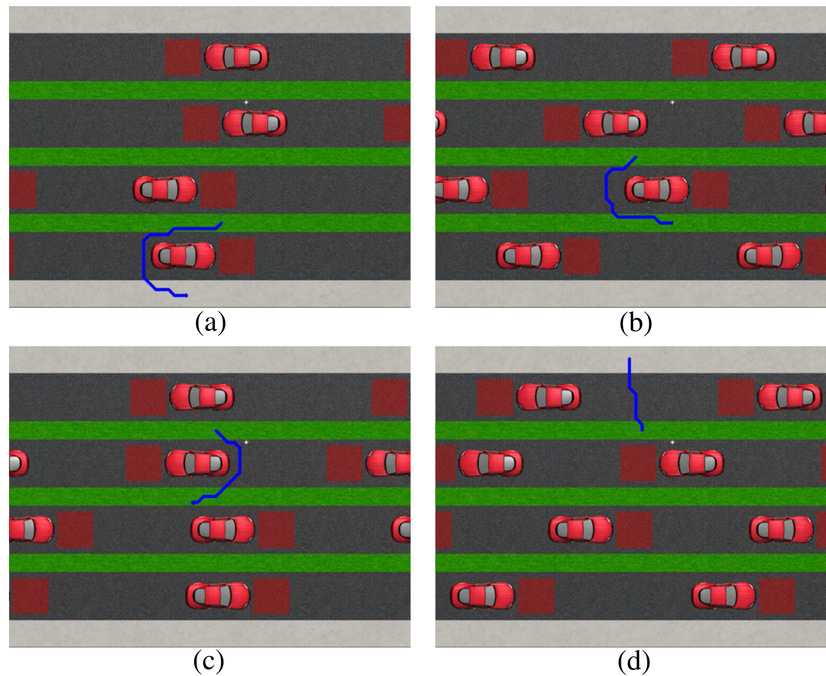


Figure 6. Navigation under different constraint specifications: dynamic constraints used to avoid navigating in front of vehicles as a proof of robustness.

stating and dynamic constraints. It is published at the following URL and is also available by request to the authors: `https://bitbucket.org/kainino/constraint-aware-navigation`.

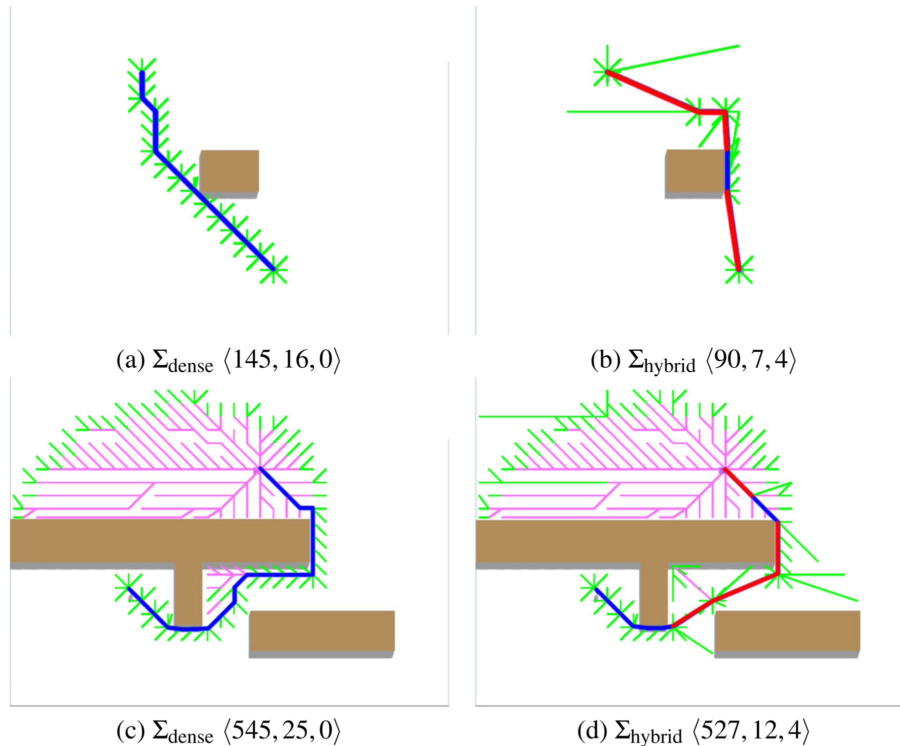
Our framework meets strict time guarantees by publishing sub-optimal paths within time constraints (e.g., within the span of one frame) and iteratively refines the plan in subsequent frames while interleaving path planning and plan execution. In practice, our framework is able to converge to an optimal path within a few frames, and additional planning time is only needed to handle dynamic constraints and goal changes. For large changes that invalidate the current path, the AD* algorithm repairs the solution efficiently by increasing the inflation factor (temporarily trading optimality for computational efficiency) and refining the plan as soon as time permits.

7.1. Evaluation of Hybrid Domain

The node expansion count (correlated with the cost depth of the search) dominates the computational complexity of the search. The use of *highway* transitions (transitions from $\mathbf{A}_{\text{tri}}$) reduces the search depth, as the length of a utilized transition from $\mathbf{A}_{\text{tri}}$ is, on average, much longer than a transition in $\mathbf{A}_{\text{dense}}$ (depending on the triangulation method and the grid density, respectively, in our examples, around 2–6 times longer). Figure 7 compares the use of Σ_{hybrid} and Σ_{dense} for the same problem instance. We observe that there is a reduction from 145 to 90 nodes expanded for just four highway nodes used in the plan for the small problem instance in Figures 7(a,b). The problem instance in Figure 7(c,d) is particularly challenging for the planner as the heuristic focuses the search in directions that are ultimately blocked. This leads to a significantly greater exploration of nodes in Σ_{dense} before a solution can be found and dilutes the benefits of highway selection. We see a reduction from 545 to 527 nodes.

Based on our experiments, we observe that the number of highway nodes n_h used in the final plan reduces the number of nodes expanded in the search by around $10n_h$ nodes in small environments. During suboptimal planning (with an inflation factor greater than one), well-aligned highway transitions can help guide the planner to a suboptimal solution. This varies depending upon the environment configuration, the number and type of constraints used, and where in the plan a highway node is chosen (The earlier a highway node is chosen during plan computation, the more significant its impact on the reduction in node expansion.).

Highway selection. The selection of highway nodes depends on the quality of triangulation, and the relative position of the start and goal, in comparison with where these nodes are present in the environment. This could be potentially mitigated by using navigation meshes with different qualities [17–22]. The inflation factor used in the search also influences highway selection. For a high inflation factor, the search is more prone to selecting highway nodes because they accelerate and focus the search while compromising optimality of solution.



(a) $\Sigma_{\text{dense}}\ \langle 145, 16, 0\rangle$ (b) $\Sigma_{\text{hybrid}}\ \langle 90, 7, 4\rangle$

(c) $\Sigma_{\text{dense}}\ \langle 545, 25, 0\rangle$ (d) $\Sigma_{\text{hybrid}}\ \langle 527, 12, 4\rangle$

Figure 7. Comparative evaluation of dense and hybrid domains. Blue indicates transitions in A_{dense}, and red indicates highway transitions from A_{tri}. Numbers shown are ⟨number of nodes expanded, number of dense nodes chosen in path, and number of highway nodes chosen in path⟩ (Some minor issues with the connectivity of the graph, due to nearby obstacles, cause some localized suboptimality in the path.). (a) Σ_{dense} ⟨145,16,0⟩; (b) Σ_{hybrid} ⟨90,7,4⟨; (c) Σ_{dense} ⟨545,25,0⟨; and (d) Σ_{hybrid} ⟨527,12,4⟩.

7.2. Evaluation of Adaptive Highway Domain

The adaptive highway domain works independently of any triangulated representation or other highly subjective input, instead lazily pruning transitions to reduce the number of node expansions and improve computational efficiency. This has advantages due to the close dependence of the domain upon the constraint system, which allows more intelligent expansion optimizations. In addition, it can even be used in world models with no high-quality triangulation available.

Even in a highly constrained planning problem (Figure 2(a,b)), the state expansions are still significantly reduced. Here, the planner expands 892 states in the dense uniform domain but only 637 states when using the adaptive highway domain; this is a 30% improvement, even in a very constraint-dense environment. In the absence of all constraints (Figure 2(c,d)), state expansion is improved from 493 states to 367 states, a 25% improvement, showing that the performance improvement is not very dependent upon the number of constraints. This is due to the adaptive highway domain's awareness of the constraint system: its performance improvement is generally more significant in the presence of constraints.

Figures 8, 9, and 10 show plots of planner results over the amount of time spent planning. This is independent of the amount of time given to the planner to plan each frame.[‡] Planning time measurements are summarized in Table I.

This data show a significant performance improvement when using the adaptive highway domain, despite a small reduction in path optimality. Although the total planning time for A* is better than AD*, note that the relative planning time for A* is much longer than AD*, so re-planning with A* is still impractically slow, especially in larger environments.

7.3. Constraint System Examples

Simple examples. Figure 4 illustrates a variety of navigation examples for a simple environment. Static obstacles and agents are annotated to add additional nodes in the triangulation to accommodate spatial relationships

[‡]Depending on the importance of the plan results, more or less time can be given to the planner to plan; for example, an autonomous robotic agent might dedicate a full CPU core to planning while a mobile game might try to minimize time spent planning.

 Comp. Anim. Virtual Worlds 2015; **26**:119–139 © 2014 John Wiley & Sons, Ltd.
DOI: 10.1002/cav

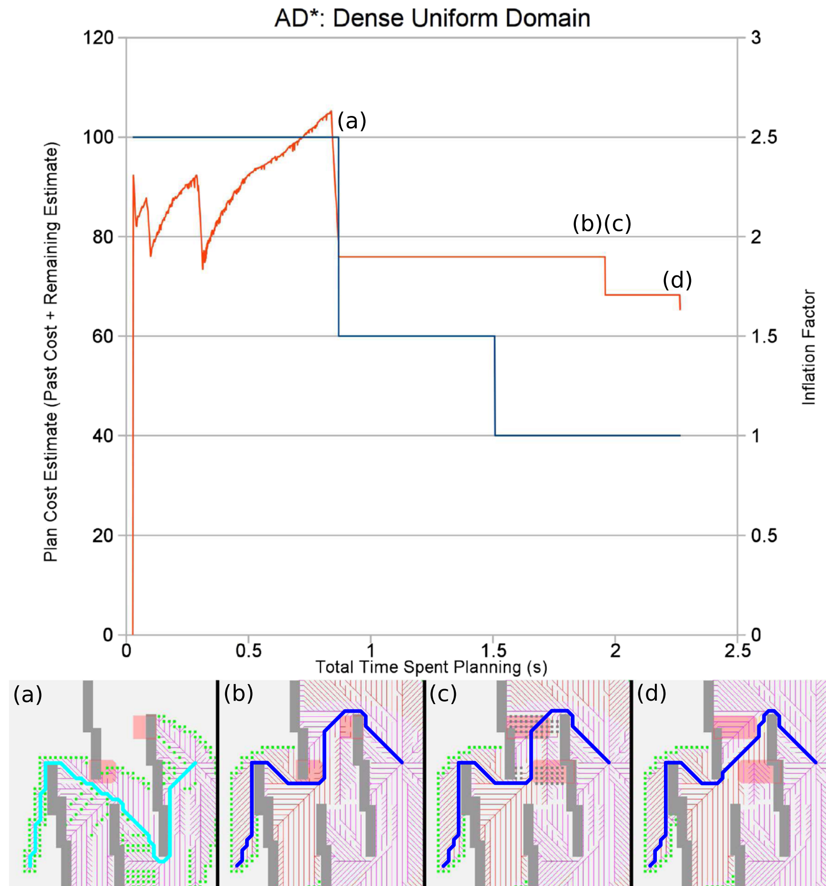


Figure 8. Plan cost versus time plot for dense uniform domain. Planning is rather time intensive but very accurate. Partial solutions are provided until (a) a suboptimal solution is found. (b) An optimal solution is found. (c) The constraints mutate, producing 'invalid' states. (d) A new optimal solution is found.

including `Between`, `Front`, `Back`, and `Left`. The hybrid graph, illustrated in Figure 4(b), combines the transitions in Σ_{tri} and Σ_{dense}. A specific problem instance **P**, illustrated in 4c, includes a start, goal configuration, and a set of hard and/or soft constraints. In this example, the agent is instructed to go `Near Between` B and C (a soft attractor), `Not Near LineOfSight` of the agent (a soft repeller), and `Not In` the grass (a well-defined area with a soft repeller). Figure 4(e) illustrates the resulting node expansion and path produced, which is drastically different from the static optimal path without any constraints, shown in Figure 4(d). Figure 4(f,g) illustrates the efficient plan repair to accommodate constraint changes, where the plan must be refined to avoid the line of sight of a moving agent. By changing the relative influence of the constraints using constraint weights w, we can produce different results where one constraint gains priority over another. In this example, the constraint to avoid line of sight is stronger than the constraint to stay between the two obstacles. Hence, we observe that if no valid path exists that satisfies all constraints, a solution is produced that accommodates as many constraints as possible, based on weights. Figure 4(h) illustrates multiple agents planning with different combinations of constraints.

Game environments. We also demonstrate our framework on challenging game environments [48]. The method of constraint specification using simple prepositional phrases is extensible, and simple atomic constraints can be easily combined to create more complex, composite constraints. Compound constraints such as staying along the wall or alternating between the left and right of obstacles to produce a zigzag path can be created by using combinations of multiple attractors and repellers, as shown in Figure 11(a–c). Figure 11(d) illustrates multiple agents conforming to a common set of constraints in their paths, emulating a lane formation or single-file behavior (However, our method is *not* suitable to crowd simulation, as there is a high computational and memory cost to run one instance of the planner. It is intended for use with a small number of agents which must navigate intelligently.).

Figure 6 shows the use of constraints in a road-crossing scenario, where the agent avoids navigating in front of moving vehicles. Although this demo does not show a very practical use of constraint planning, it illustrates

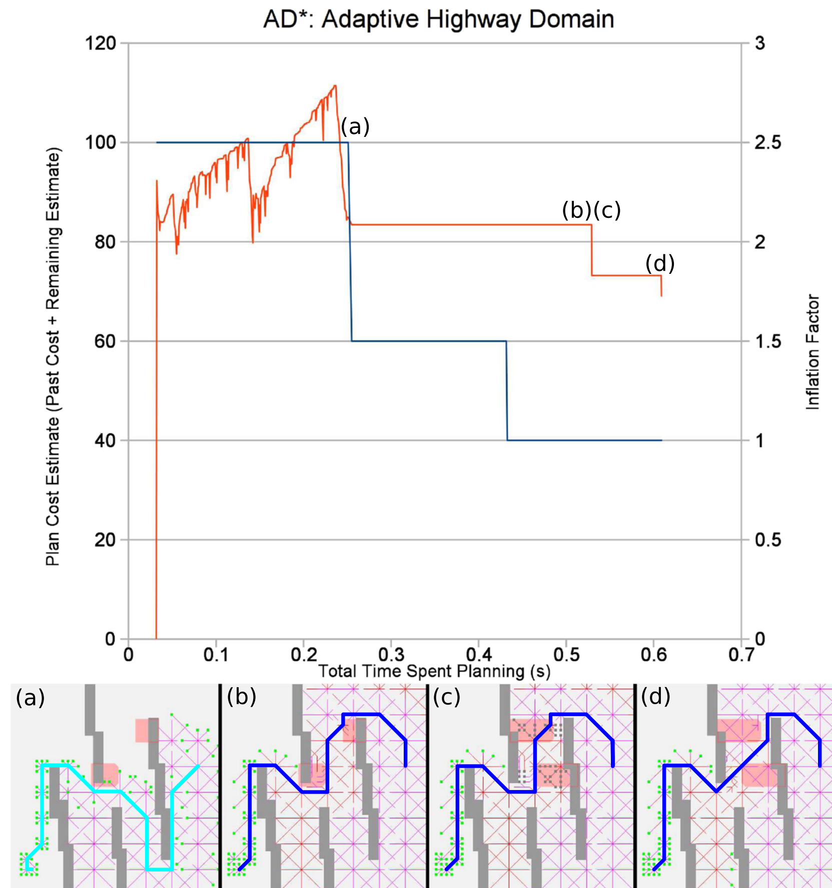


Figure 9. Plan cost versus time plot for adaptive highway domain. Total planning time is found to be much shorter than the dense uniform domain at the cost of a small amount of plan accuracy. Partial solutions are provided until (a) a suboptimal solution is found. (b) An optimal solution is found. (c) The constraints mutate, producing 'invalid' states. (d) A new optimal solution is found.

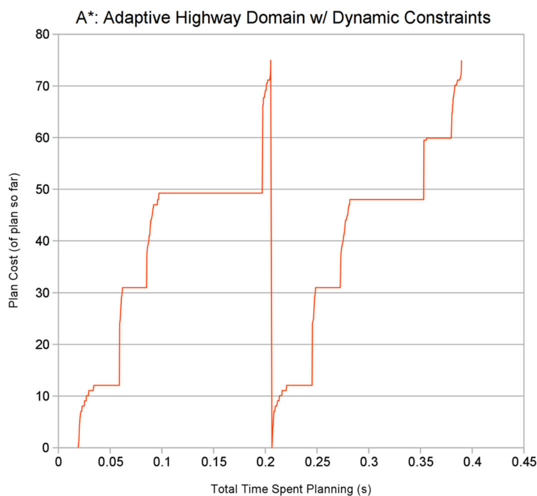


Figure 10. Plan cost versus time plot for A* in the adaptive highway domain. The same scenario is shown, without dynamic plan repairs. Instead, A* is run again from scratch after the constraints are modified.

Table I. Planning time measurements (total sum of program time spent planning) with various domains and planners.

Plan time/plan cost	AD*, dense	AD*, adaptive	A*, adaptive
To suboptimal plan	0.84 s/76	0.22 s/83	—
To optimal plan	1.94 s/68	0.50 s/73	0.19 s
Plan repair	0.33 s/65	0.04 s/69	0.18 s

considerable robustness in such a rapidly changing environment even without any awareness of obstacle trajectory (Future work in this area is discussed under Future Work in Section 8.).

Plan repair to avoid the line of sight of multiple moving agents is shown in Figure 12. Here, the user interactively selects agents associated with the constraints and changes their position, thus invalidating the current plan. The same problem configuration using attractor constraints for `LineOfSight` produces a drastic change in the resulting path, as shown in Figure 13. Our framework efficiently repairs the existing solution to accommodate the constraint changes.

Human relationship constraints. The work of Sturtevant [45] incorporates human relationships (personal dis-

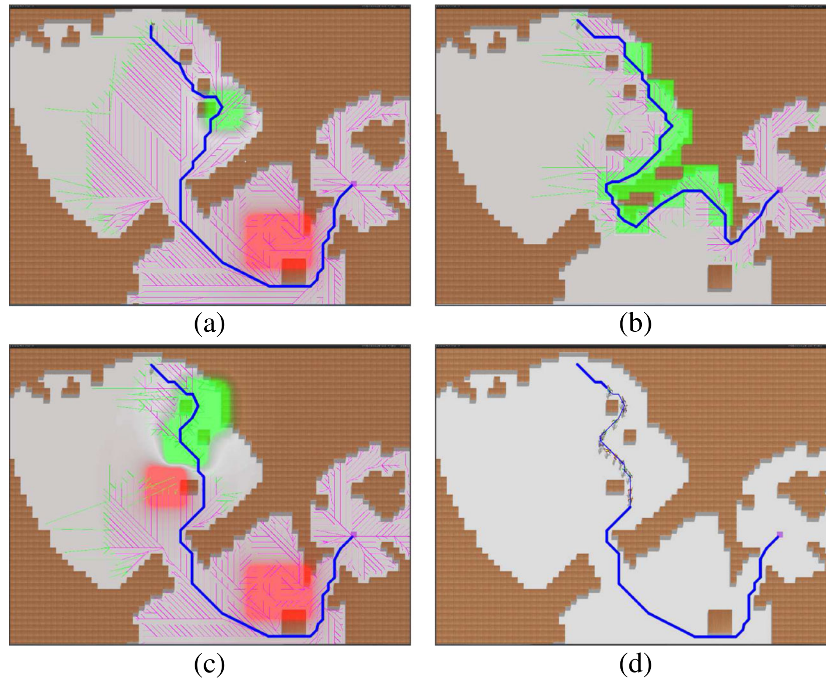


Figure 11. Navigation under different constraint specifications. (a) Attractor to go behind an obstacle and a repeller to avoid going in front of an obstacle. (b) Combination of attractors to go along a wall. (c) Combination of attractors and repellers to alternate going in front of and behind obstacles, producing a zigzag trajectory. (d) Emulating lane formation using multiple agents under the same constraints.

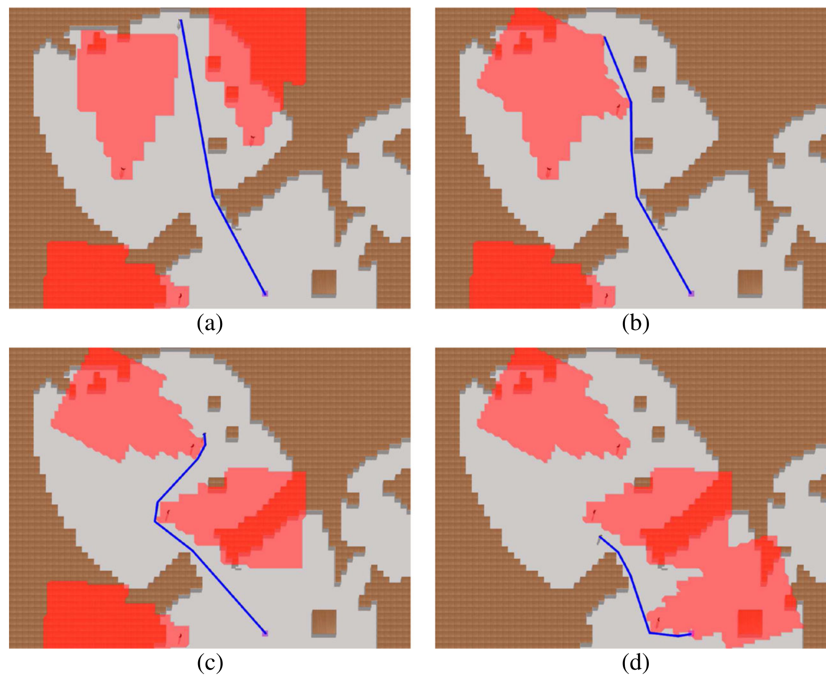


Figure 12. `Not In LineOfSight` constraint. Utilizes multiple dynamic agents. A user interactively moves agents and the plan is repaired to accommodate constraint change.

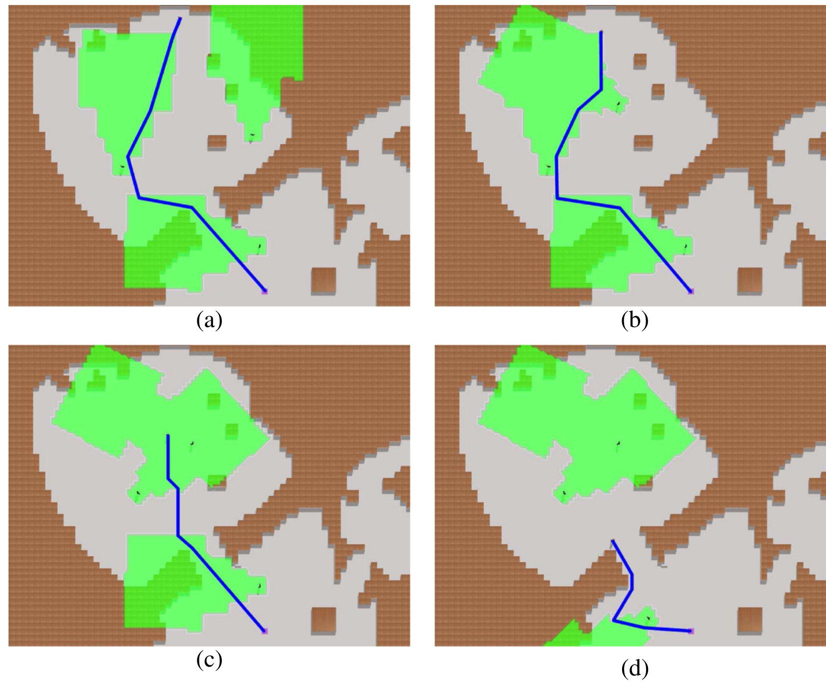


Figure 13. In LineOfSight constraint. Utilizes multiple dynamic agents. A user interactively moves agents and the plan is repaired to accommodate constraint change.

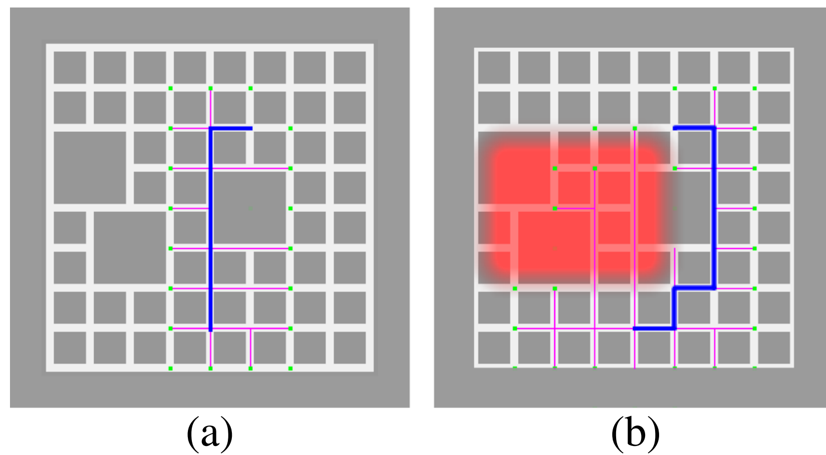


Figure 14. Demonstration of our framework applied to a map navigation problem using examples in a city grid layout. (a) A simple no-constraint solution. (b) Addition of a Not Near NeighborhoodX constraint on the same map. For example, the negative constraint might be used at night to avoid a dangerous neighborhood, but during the day, that constraint may be removed or weight-reduced.

tance and line of sight) into path planning using a similar approach to our framework. In our previous work [46], we used a rough approximation of line of sight to demonstrate dynamic constraints. In the updated framework, we use a more robust human relationship constraint modeled after Sturtevant's human relationship constraints. Figure 3 shows a simple environment that uses this constraint.

Large-scale environments. Although most of our examples operate at human movement scales, our framework can operate at other scales, such as map navigation. This can be useful not only for autonomous agent planning but also for automatic route planning. In Figure 14, a simple neighborhood-avoidance problem is demonstrated, which operates in a grid-based city layout.

Spatiotemporal constraint specification. Another type of problem we have briefly explored, to illustrate support for other useful tactical behavior specifications, is that of time-based and date-based constraints. Because the

framework allows for fully dynamic constraints, it is possible to create spatial constraints, which change weight (or disappear) before or during execution, depending on some varying input such as time of day or other agent/environment status (e.g., 'Are the guards alerted? If so, avoid some regions'.). In Figure 14, we show a planning problem with varying weight in order to avoid a certain neighborhood at night.

7.4. Parameter Selection and Performance

For our experiments, ϵ was initially set to a value of 2.5 to quickly produce a sub-optimal solution while meeting time constraints, which could be iteratively refined over subsequent plan iterations. $t_{\max}$ was set to 0.032 s and the plan computations of multiple agents were distributed over successive frames to ensure that the frame rate was always greater than 30 Hz. The maximum allotted time can be further calibrated to introduce limits on computational resources or accommodate many characters at the expense of plan quality. We observed that in general, the value of ϵ quickly converges to 1.0 to produce an optimal path and requires only a few frames to repair solutions to accommodate dynamic events (Table I). For rapid changes in the environment over many frames, the planner may be unable to find a solution and the agent stops moving till a valid path is computed for execution.

The AD* algorithm requires all visited nodes in the search graph to be cached to facilitate efficient plan repair, imposing a memory overhead for large environments. There exists a trade-off between computational performance and memory requirements, where using a traditional A* search would require less nodes to be stored at the expense of planning from scratch whenever the plan is invalidated.

The choice of the base multiplier m_0 impacts how constraints affect the resulting cost formulation, with higher values diluting the influence of the distance cost and the heuristic on the resulting search. We automatically pick the lowest possible value of m_0 to accommodate the maximum value of attractor constraints while preserving optimality guarantees. A cost model where the base multiplier has no adverse effect on admissibility or the influence of the heuristic is the subject of future work.

8. CONCLUSION

We present a goal-directed navigation system that satisfies multiple spatial constraints imposed on the path. Constraints can be specified with respect to obstacles in the environment, as well as other agents. For example, a path to a target could be altered to stay behind buildings and walk along walls while avoiding line of sight with patrolling guards. An extended anytime dynamic planner is used to compute constraint-aware paths while efficiently repairing solutions to account for dynamic constraints.

Future work. The performance of the hybrid domain is sensitive to the kind of triangulations produced for the environment. For future work, we would like to explore better automated triangulation solutions [17–22] and manually annotated waypoint graphs to improve computational performance. The performance improvements of the adaptive highway domain are contingent upon the constraint weights and their area of influence, which need to be addressed to provide an efficient discrete environment representation that generalizes across different constraints. In addition, we would like to explore solutions to the limitations mentioned in Section 4.4. Static analysis of the environment could potentially yield automatic annotation generation (e.g., `Between`, `Inside`), which would improve ease of creating spatial relationships and is a subject of future exploration. In addition, our ahead-of-time planning could be extended to include trajectory extrapolation by using modified annotation positions based on search depth; this could improve plans in environments such as the road-crossing example. Finally, we have most closely considered spatial constraints in this paper, but our framework is general and extensible to other problem domains. We would like to expand upon other types of planning problems. Although we briefly explored spatiotemporal constraints, we see many possible ways to expand this to more advanced problems.

To highlight the benefits of our method, all constraints were accounted for at the global planning layer. However, in some cases where the constraint is constantly changing, such as a moving vehicle, it may be significantly more efficient to use a locally optimal strategy for constraint satisfaction. A hybrid approach that combines the benefits of both global planning and local collision avoidance for constraint satisfaction is another subject of future exploration.

ACKNOWLEDGEMENTS

The research reported in this document/presentation was performed in connection with Contract Number W911NF-10-2-0016 with the U.S. Army Research Laboratory. The views and conclusions contained in this document are those of the authors and should not be interpreted as presenting the official policies or position, either expressed or implied, of the U.S. Army Research Laboratory, or the U.S. Government. Citation of manufacturers or trade names does not constitute an official endorsement or approval of the use thereof. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon.

APPENDIX A: ANYTIME DYNAMIC PLANNER

EventHandler (Algorithm 3 [1–8]) monitors events in the simulation and triggers appropriate routines. **ComputeOrImprovePath** (Algorithm 2 [15–23]) is invoked each

Algorithm 2 Anytime dynamic planner (Part 1)

1: **function KEY**s
2: **if** $g(s) > rhs(s)$ **then**
3: **return** $[rhs(s) + \epsilon \cdot \mathrm{h}(s, s_{\mathrm{goal}}); rhs(s)]$
4: **elsereturn** $[g(s) + \cdot \mathrm{h}(s, s_{\mathrm{goal}}); g(s)]$

5: **function UPDATESTATE**s
6: **if** $(s \neq s_{\mathrm{start}})$ **then**
7: $s' = \arg_{s' \in \mathtt{pred}(s)} \min(c(s, s') \cdot M_{\mathbf{C}}(s, s') + g(s'))$
8: $rhs(s) = c(s, s') \cdot M_{\mathbf{C}}(s, s') + g(s')$
9: $prev(s) = s'$
10: **if** ($s \in$ OPEN) remove s from OPEN
11: **if** $g(s) \neq rhs(s)$ **then**
12: **if** ($s \notin$ CLOSED) insert s in OPEN with key(s)
13: **else** insert s in INCONS
14: Insert s in VISITED

15: **function COMPUTEORIMPROVEPATH**$(t_{\max})$
16: **while** $(\min_{s \in \mathtt{OPEN}}(key(s) < key(s_{\mathrm{goal}}) \vee rhs(s_{\mathrm{goal}}) \neq g(s_{\mathrm{goal}}) \vee \Pi(s_{\mathrm{start}}, s_{\mathrm{goal}}) = \mathtt{NULL}) \wedge t < t_{\max}$ **do**
17: $s = \arg_{s \in \mathtt{OPEN}} \min(key(s))$
18: **if** $(g(s) > rhs(s))$ **then**
19: $g(s) = rhs(s)$
20: $\mathtt{CLOSED} = \mathtt{CLOSED} \cup s$
21: **else**
22: $g(s) = \infty$
23: **UpdateState**s

time the planning task is executed. This function monitors events and calls the appropriate event handlers for changes in start, goal, and constraints. Given a maximum amount to deliberate $t_{\max}$, it refines the plan and publishes the ϵ-suboptimal solution using the AD* planning Algorithm [27]. We briefly describe our implementation of the AD* algorithm and how we handle changes in start, goal, and constraint movement and refer the readers to the work of Likhachev *et al.* [27] for more details.

AD* performs a backward search and maintains a least cost path from the goal s_{goal} to the start s_{start} by storing the cost estimate $g(s)$ from s to s_{goal}. However, in dynamic environments, edge costs in the search graph may constantly change and expanded nodes may become inconsistent. Hence, a one-step look ahead cost estimate $rhs(s)$ is introduced [25] to determine node consistency.

The priority queue OPEN contains the states that need to be expanded for every plan iteration, with the priority defined using a lexicographic ordering of a two-tuple **key**s, defined for each state. OPEN contains only the inconsistent states ($g(s) \neq rhs(s)$) that need to be updated to become consistent. Nodes are expanded in increasing priority until there is no state with a key value less than the start state. A heuristic function $h(s, s')$ computes an estimate of the optimal cost between two states and is used to focus the search toward s_{start}.

Instead of processing all inconsistent nodes, only those nodes whose costs may be inconsistent beyond a certain bound, defined by the inflation factor ϵ are expanded. It performs an initial search with an inflation factor ϵ_0 and is guaranteed to expand each state only once. An INCONS list keeps track of the already expanded nodes that become inconsistent because of cost changes in neighboring nodes. Assuming no world changes, ϵ is decreased iteratively and plan quality is improved until an optimal solution is reached ($\epsilon = 1$). Each time ϵ is decreased, all states made inconsistent because of change in ϵ are moved from INCONS to OPEN with **key**s based on the reduced inflation factor, and CLOSED is made empty. This improves efficiency because it only expands a state at most once in a given search, and reconsidering the states from the previous search that were inconsistent allows much of the previous search effort to be reused, requiring only a minor amount of computation to refine the solution. **ComputeOrImprovePath** (Algorithm 2 [15–23]) gives the routine for computing or refining a path from s_{start} to s_{goal}.

When change in edge costs are detected, new inconsistent nodes are placed into OPEN and node expansion is repeated until a least cost solution is achieved within the current ϵ*bounds*. When the environment changes substantially, it may not be feasible to repair the current solution and it is better to increase ϵ so that a less optimal solution is reached more quickly.

An increase in edge cost may cause states to become under-consistent ($g(s) < rhs(s)$) where states need to be inserted into OPEN with a key value reflecting the minimum of their old and new costs. In order to guarantee that under-consistent states propagate their new costs to their affected neighbors, their key values must use uninflated heuristic values. This means that different key values must be computed for under-consistent and over-consistent states, as shown in Algorithm 2 [1–4]. This key definition allows AD* to efficiently handle changes in edge costs and changes to inflation factor.

AD* uses a backward search to handle agent movement along the plan by recalculating key values to automatically focus the search repair near the updated agent state. It can handle changes in edge costs due to obstacle and start movement and needs to plan from scratch each time the goal changes. The routines to handle start and goal changes are described, whereas the routine to handle constraint changes is described in Algorithm 1.

StartChangeUpdate. When the start moves along the current plan, the key values of all states in OPEN are recomputed to re-prioritize the nodes to be expanded. This focuses processing toward the updated agent state allowing the agent to improve and update its solution path while it is being traversed. When the new start state deviates substantially from the path, it is better to plan from scratch. Algorithm 3 [9–16] provides the routine to handle start movement.

GoalChangeUpdate. Algorithm 3 [17–20] clears plan data and resets ϵ whenever the goal changes and plans from scratch at the next step.

Algorithm 3 Anytime dynamic planner (Part 2)

1: **function** EVENTHANDLER
2: 　**if** `START_CHANGED` **then**
3: 　　**StartChangeUpdate**(s_c)
4: 　**if** `GOAL_CHANGED` **then**
5: 　　**GoalChangeUpdate**(s_{new})
6: 　**if** `CONSTRAINT_CHANGED` **then**
7: 　　**for all** constraint change c **do**
8: 　　　**ConstraintChangeUpdate**($c, \vec{x}_{\text{prev}}, \vec{x}_{\text{next}}$)

9: **function** STARTCHANGEUPDATE()s_c
10: 　**if** $s_c \notin \Pi(s_{\text{start}}, s_{\text{goal}})$ **then**
11: 　　**ClearPlanData**()
12: 　　$\epsilon = \epsilon_0$
13: 　**else**
14: 　　$s_{\text{start}} = s_c$
15: 　　**for all** $s \in$ `OPEN` **do**
16: 　　　Update **key**s

17: **function** GOALCHANGEUPDATE(s_{new})
18: 　**ClearPlanData**()
19: 　$\epsilon = \epsilon_0$
20: 　$s_{\text{goal}} = s_{new}$

REFERENCES

1. Kapadia M, Badler NI. Navigation and steering for autonomous virtual humans. *Wiley Interdisciplinary Reviews: Cognitive Science* 2013; **4**(3): 263–272.
2. Al Marzouqi M, Jarvis RA. Robotic covert path planning: A survey. In *Proceedings of the 2011 IEEE Conference on Robotics, Automation and Mechatronics (RAM)*, Qingdao, China, 2011; 77–82.
3. Warren CW. Global path planning using artificial potential fields. In *Proceedings of the 1989 IEEE International Conference on Robotics and Automation, volume 1*, Scottsdale, Arizona, USA, 1989; 316–321.
4. Warren CW. Multiple robot path coordination using artificial potential fields. In *Proceedings of the 1990 IEEE International Conference on Robotics and Automation,* volume 1, Cincinnati, Ohio, USA, 1990; 500–505.
5. Shimoda S, Kuroda Y, Iagnemma K. Potential field navigation of high speed unmanned ground vehicles on uneven terrain. In *Proceedings of the 2005 IEEE International Conference on Robotics and Automation*, Barcelona, Spain, 2005; 2828–2833.
6. Goldenstein S, Karavelas M, Metaxas D, Guibas L, Aaron E, Goswami A. Scalable nonlinear dynamical systems for agent steering and crowd simulation 2001; **25**(6): 983–998.
7. Arkin RC. Motor schema based navigation for a mobile robot: an approach to programming by behavior. In *Proceedings of the 1987 IEEE International Conference on Robotics and Automation,* volume 4, Raleigh, North Carolina, USA, March 1987; 264–271.
8. Treuille A, Cooper S, Popović Z. Continuum crowds. *ACM Transactions on Graphics* 2006; **25**(3): 1160–1168.
9. Kapadia M, Singh S, Hewlett W, Faloutsos P. Egocentric affordance fields in pedestrian steering. In *Proceedings of the 2009 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. I3D'09. ACM: New York, NY, USA, 2009; 215–223.
10. Kapadia M, Singh S, Hewlett W, Reinman G, Faloutsos P. Parallelized egocentric fields for autonomous navigation. *The Visual Computer* 2012; **28**: 1–19. DOI: 10.1007/s00371-011-0669-5.
11. Mitchell JSB, Papadimitriou CH. The weighted region problem: finding shortest paths through a weighted planar subdivision. *Journal of the ACM* January 1991; **38**(1): 18–73.
12. Narayanappa S. and University of Denver. *Exact solutions for simple weighted region problems*. University of Denver, 2006. http://books.google.ch/books?id=jAoJV750QcIC.
13. Reif JH, Sun Z. An efficient approximation algorithm for weighted region shortest path problem. In *Algorithmic and Computational Robotics: New Directions*, Donald BR, Lynch KM, Rus D (eds). A.K. Peters: Wellesley, MA, 2001; 191–203.
14. Hart PE, Nilsson NJ, Raphael B. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics* 1968; **4**(2): 100–107.
15. Hart PE, Nilsson NJ, Raphael B. Correction to "A formal basis for the heuristic determination of minimum cost paths". *SIGART Bulletin* 1972; **37**: 28–29.
16. Dechter R, Pearl J. Generalized best-first search strategies and the optimality of A*. *Journal of the ACM* 1985; **32**(3): 505–536.
17. Mononen M. Recast: navigation-mesh construction toolset for games, 2009. http://code.google.com/p/recastnavigation/.
18. Kallmann M. Shortest paths with arbitrary clearance from navigation meshes. In *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. SCA'10. Eurhics

Association: Aire-la-Ville, Switzerland, Switzerland; 159–168. http://dl.acm.org/citation.cfm?id=1921427.1921451, 2010.

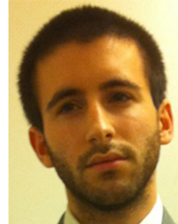
19. Oliva R, Pelechano N. Automatic generation of sub-optimal navmeshes. In *Proceedings of the 2011 International Conference on Motion in Games (MIG)*. MIG'11. Springer-Verlag: Berlin, Heidelberg, 2011; 328–339.
20. Oliva R, Pelechano N. NEOGEN: near optimal generator of navigation meshes for 3D multi-layered environments. *Computers & Graphics* 2013; **37**(5): 403–412.
21. Pettré J, Laumond JP, Thalmann D. A navigation graph for real-time crowd animation on multilayered and uneven terrain. *First International Workshop on Crowd Simulation* 2005; **43**(44): 194.
22. van Toll WG, Cook AF, Geraerts R. A navigation mesh for dynamic environments. *Computer Animation and Virtual Worlds* November 2012; **23**(6): 535–546.
23. Schultes D. *Route Planning in Road Networks*. VDM Verlag: Saarbrücken, Germany, Germany, 2008.
24. Geisberger R, Sanders P, Schultes D, Delling D. Contraction hierarchies: faster and simpler hierarchical routing in road networks. In *Proceedings of the 7th International Conference on Experimental Algorithms*. WEA'08. Springer-Verlag: Berlin, Heidelberg, 2008; 319–333.
25. Koenig S, Likhachev M. D* Lite. In *Proceedings of the 2002 National Conference on Artificial Intelligence*. AAAI: Menlo Park, CA, USA, 2002; 476–483.
26. Likhachev M, Gordon GJ, Thrun S. Ara*: anytime A* with provable bounds on sub-optimality. In *Proceedings of the 2003 Conference on Advances in Neural Information Processing Systems*, Vancouver, B.C., 2003, Vol. 16, 767–774.
27. Likhachev M, Ferguson DI, Gordon GJ, Stentz A, Thrun S. Anytime dynamic A*: an anytime, replanning algorithm. In *Proceedings of the 2005 International Conference on Automated Planning and Scheduling*, Monterey, California, USA, 2005; 262–271.
28. Pelechano N, Allbeck JM, Badler NI. *Virtual Crowds: Methods, Simulation, and Control*, Synthesis Lectures on Computer Graphics and Animation. Morgan & Claypool Publishers: San Rafael, California, 2008.
29. Reynolds C. Steering behaviors for autonomous characters. In *Proceedings of the 1999 Game Developers Conference*, San Jose, California, 1999; 763–782.
30. Helbing D, Molnár P. Social force model for pedestrian dynamics. *Physical Review E* May 1995; **51**(5): 4282–4286.
31. Pelechano N, Allbeck JM, Badler NI. Controlling individual agents in high-density crowd simulation. In *Proceedings of the 2007 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. SCA'07. Eurographics Association: Aire-la-Ville, Switzerland, Switzerland, 2007; 99–108.
32. Paris S, Pettré J, Donikian S. Pedestrian reactive navigation for crowd simulation: a predictive approach. *Computer Graphics Forum* 2007; **26**(3): 665–674.
33. van den Berg J, Lin MC, Manocha D. Reciprocal velocity obstacles for real-time multi-agent navigation. In *Proceedings of the 2008 IEEE International Conference on Robotics and Automation*. IEEE: Pasadena, CA, 2008; 1928–1935.
34. Singh S, Kapadia M, Hewlett B, Reinman G, Faloutsos P. A modular framework for adaptive agent-based steering. In *Proceedings of the 2011 Symposium on Interactive 3D Graphics and Games*. I3D'11. ACM: New York, NY, USA, 2011; 141–150.
35. Singh S, Kapadia M, Reinman G, Faloutsos P. Footstep navigation for dynamic crowds. *Computer Animation and Virtual Worlds* 2011; **22**(2-3): 151–158.
36. Kapadia M, Beacco A, Garcia F, Reddy V, Pelechano N, Badler NI. Multi-domain real-time planning in dynamic environments. In *Proceedings of the 2013 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. SCA'13. ACM: New York, NY, USA, 2013; 115–124.
37. Schuerman M, Singh S, Kapadia M, Faloutsos P. Situation agents: agent-based externalized steering logic. *Computer Animation and Virtual Worlds* May 2010; **21**: 267–276.
38. Xu YD, Badler NI. Algorithms for generating motion trajectories described by prepositions. In *Proceedings of Computer Animation 2000*, Philadelphia, Pennsylvania, USA, 2000; 30–35.
39. André E, Bosch G, Herzog G, Rist T. Characterizing trajectories of moving objects using natural language path descriptions. In *Proceedings of the 7th European Conference on Artificial Intelligence*, Brighton, UK, 1986; 1–8.
40. Bhattacharya S, Likhachev M, Kumar V. Search-based path planning with homotopy class constraints in 3D. In *Proceedings of the 26th AAAI Conference on Artificial Intelligence*, Toronto, Ontario, Canada, 2012; 2097–2099.
41. Bhattacharya S, Likhachev M, Kumar V. Topological constraints in search-based robot path planning. *Autonomous Robots* 2012; **33**(3): 273–290.
42. Hernandez E, Carreras M, Galceran E, Ridao P. Path planning with homotopy class constraints on bathymetric maps. In *Oceans - Europe*, 2011; 1–6.
43. Phillips M, Hwang V, Chitta S, Likhachev M. Learning to plan for constrained manipulation from demonstrations. In *Robotics: Science and Systems IX*. Technische Universität Berlin: Berlin, Germany, 2013.

44. Geraerts R. Planning short paths with clearance using explicit corridors. In *Proceedings of the 2010 IEEE International Conference on Robotics and Automation*. IEEE: Pasadena, CA, 2010; 1997–2004.
45. Sturtevant NR. Incorporating human relationships into path planning. In *Proceedings of the 9th AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, Northeastern University, Boston, Massachusetts, USA, 2013; 177–183.
46. Kapadia M, Ninomiya K, Shoulson A, Garcia F, Badler NI. Constraint-aware navigation in dynamic environments. In *Proceedings of the 2013 ACM SIGGRAPH Conference on Motion in Games (MIG)*. MIG'13. ACM: New York, USA, 2013; 111–120.
47. Shoulson A, Marshak N, Kapadia M, Badler NI. Adapt: the agent development and prototyping testbed. In *Proceedings of the 2013 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. I3D'13. ACM: New York, NY, USA, 2013; 9–18.
48. Sturtevant NR. Benchmarks for grid-based pathfinding. *Transactions on Computational Intelligence and AI in Games* 2012; **4**(2): 144–148.

AUTHORS' BIOGRAPHIES

Kai Ninomiya is an undergraduate student in computer science and master's degree student in computer graphics at the University of Pennsylvania. Kai worked on the original Motion in Games publication in the summer of 2013 as an undergraduate research assistant in the UPenn Center for Human Modeling and Simulation.

Mubbasir Kapadia is an Associate Research Scientist at Disney Research Zurich. Previously, he was a postdoctoral researcher and Assistant Director at the Center for Human Modeling and Simulation at University of Pennsylvania. He received his PhD in Computer Science at University of California, Los Angeles.



Alexander Shoulson is Ph.D. student at the University of Pennsylvania, supervised by Dr. Norman I. Badler. His research focuses on interactive narrative, practical game AI, character animation, and behavior authoring for virtual humans.

Francisco Garcia is a MS/Ph.D student at the University of Massachusetts, Amherst. He graduated with a B.S. in Computer Science from the University of the Sciences in Philadelphia in 2012. His research interests are Artificial Intelligence, Robotics, Planning and Machine Learning.

Norman I. Badler is the Rachleff Professor of Computer and Information Science at the University of Pennsylvania. He received his PhD in Computer Science from the University of Toronto in 1975. His research involves developing software for human and group behavior modeling and animation. He is the founding Director of the SIG Center for Computer Graphics and the Center for Human Modeling and Simulation at Penn.