Constraint-Aware Navigation in Dynamic Environments

Mubbasir Kapadia* Kai Ninomiya† Alexander Shoulson‡ Francisco Garcia§
University of Pennsylvania University of Massachusetts Amherst
Norman Badler¶
University of Pennsylvania

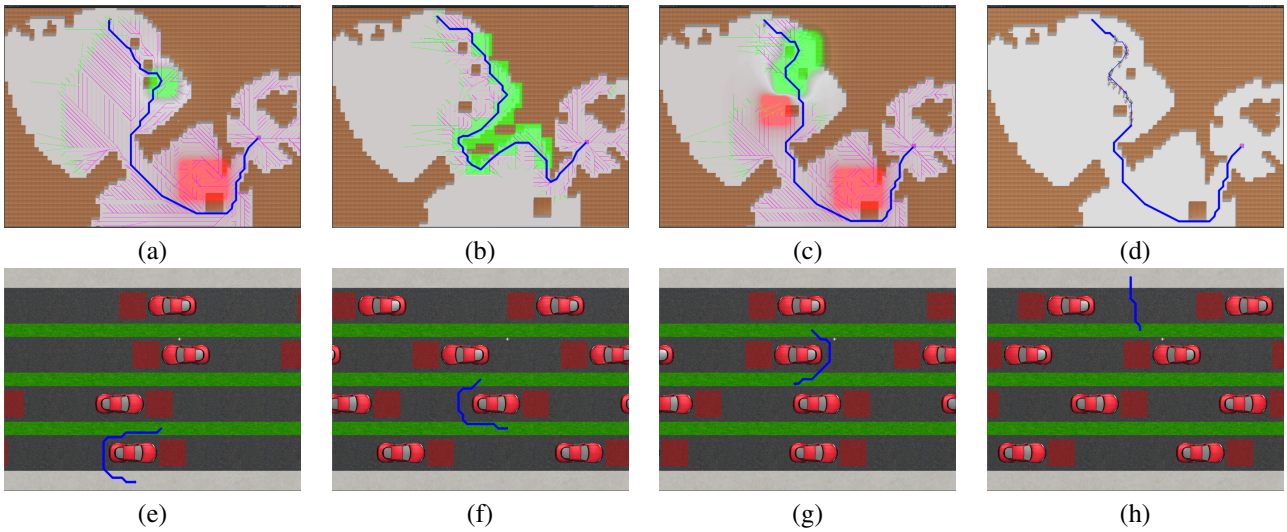


Figure 1: *Navigation under different constraint specifications. (a) Attractor to go behind an obstacle and a repeller to avoid going in front of an obstacle. (b) Combination of attractors to go along a wall. (c) Combination of attractors and repellers to alternate going in front of and behind obstacles, producing a (d) Lane formation with multiple agents under same constraints. (e)–(h) Dynamic constraints used to avoid navigating in front of vehicles.*

Abstract

Path planning is a fundamental problem in many areas ranging from robotics and artificial intelligence to computer graphics and animation. While there is extensive literature for computing optimal, collision-free paths, there is 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 could include staying behind a building, walking along walls, or avoiding the line of sight of patrolling agents. We introduce a hybrid environment representation that balances computational efficiency and discretization resolution, 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 dynamic constraints. We demonstrate the benefits of our method on challenging navigation problems in complex environments for dynamic agents using combinations of hard and soft constraints, attracting and repelling constraints, on static obstacles and moving obstacles.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Animation;

Keywords: path planning, spatial constraints, navigation, anytime dynamic planning, potential fields

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 a mental 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 [Kapadia and Badler 2013], 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 avoid-

*mubbasir.kapadia@gmail.com
†kainino@seas.upenn.edu
‡ashoulson@gmail.com
§fmaxgarcia@gmail.com
¶badler@seas.upenn.edu

ance, inter-agent relationships, and so on). This produces locally optimal results with no strict guarantees on constraint satisfaction, and does not scale well to handle constraint combinations. Computing global trajectories that account for spatial constraints with respect to obstacles and other agents is still a challenging problem [Al Marzouqi and Jarvis 2011] 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. Such constraints may include specific location investigation ("check behind the building"), dynamic agent evasion and stealth ("avoid being seen"), instructions with path requirements for the mission ("follow the path"), or organization ("stay between these two guys"). A hybrid discretization of the environment combines the computational benefits of triangulations, while a uniform grid ensures sufficient resolution to account for dynamic constraints. Hard constraints, which must be satisfied, effectively prune invalid transitions in the search graph, while soft constraints (attractors or repellers) have a multiplicative effect on the cost of choosing a transition. Constraints are represented as continuous potential 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:

- A hybrid environment representation that balances computational efficiency and discretization resolution to provide a minimal, yet sufficient discretization of the search graph for constraint-aware navigation.
- A method of constraint specification using simple prepositional phrases, that can be easily concatenated to specify complex custom constraints.
- A quantitative description of qualitative motion constraint specifications that can be applied generally to cost-minimizing pathfinding methods.
- 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 constraint changes.

To demonstrate the benefits 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 [Kapadia and Badler 2013] have been proposed for autonomous agents, some of which are described below.

Potential Fields. The approach of potential fields [Warren 1989; Warren 1990; Shimoda et al. 2005; Goldenstein et al. 2001; Arkin 1987] 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. Since a change in target or environment requires significant recomputation, these navigation methods are generally confined to systems with non-changing goals and static environments. Dynamic potential fields [Treuille et al. 2006] 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*. [2009; 2012] 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.

Discrete Graph-based Search. Discrete search methods such as A* [Hart et al. 1968; Hart et al. 1972; Dechter and Pearl 1985] 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 resolution producing low-quality paths and fine resolution grids proving to be computationally prohibitive for real-time applications. The work of Mononen [2009] and Kallmann [2010] propose a triangulation-based representation of the environment for efficient pathfinding. D* Lite [Koenig and Likhachev 2002] is able to efficiently repair computed paths to accommodate dynamic changes in the environment, while ARA* [Likhachev et al. 2003] provides anytime solution guarantees with strict bounds on suboptimality. These methods provide the basis for AD* [Likhachev et al. 2005], 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 extensive work [Pelechano et al. 2008] that relies on local goal-directed collision-avoidance for simulating large crowds. These include rule-based approaches [Reynolds 1999], social forces [Helbing and Molnár 1995; Pelechano et al. 2007], predictive methods [Paris et al. 2007; van den Berg et al. 2008; Singh et al. 2011a], local fields [Kapadia et al. 2009], and planning-based approaches [Singh et al. 2011b; Kapadia et al. 2013]. The work of Schuerman *et al*. [2010] externalizes steering logic to enforce local constraints for group formations.

Navigation with Constraints. The work of Xu and Badler [2000] describes a list of representative prepositions for constraining motion trajectories in goal-directed navigation. The work of André [André et al. 1986] analyses 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.* [2012b; 2012a] explore the use of homotopy class of trajectories in graph-based search for path planning with constraints. This method is extended by Bhattacharya et al. [2012b] to handle 3D spaces. A homotopy class-based approach to A* called *HA** [Hernandez et al. 2011], ensures optimality and directs the search by exploring areas that satisfy a given homotopy class. The work of Phillips *et al*. [2013] demonstrates constrained manipulation using experience graphs. The work in [Geraerts 2010; Kallmann 2010] embeds additional information in the underlying environment representation to efficiently compute shortest paths with clearance constraints.

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. We use a hybrid environment representation that combines the benefits of triangulations and a uniform grid, and annotate the environment to focus and accelerate the search in the constrained region. Compared to [Xu and Badler 2000], we utilize an anytime dynamic planner which 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 [André et al. 1986].

3 Problem Definition

The problem domain, $\Sigma = \langle \mathbf{S}, \mathbf{A} \rangle$ defines the set of all possible states $\mathbf{S}$, and the set of permissible transitions $\mathbf{A}$. Every problem instance $\mathbf{P}$, 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 $\mathbf{C}$ is the set of active hard and soft constraints. A hard constraint is used to prune transitions in $\mathbf{A}$. For example, consider a flower bed which *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 lead the agent towards a certain region in space or away from it. A planner generates a plan, $\Pi(s_{\text{start}}, s_{\text{goal}})$, which is a sequence of states from s_{start} to s_{goal} that satisfies $\mathbf{C}$.

The rest of this paper is organized as follows. Section 4 describes the discretization and annotation of the environment to define Σ. Section 5 discusses the specification of constraints and their representation as multiplier fields. Section 6 describes the planning algorithm for generating paths with spatial constraints in dynamic environments. Section 7 evaluates our method, with concluding remarks in Section 8.

4 Environment Representation

In this section, we describe the discretized environment representation that we use for constraint-aware pathfinding. A coarse-resolution representation facilitates efficient search, but cannot accommodate all constraints due to insufficient resolution in regions of the environment where constraints maybe specified. A dense representation of the environment can account for all constraints (including dynamic objects), but is not efficient for large environments. To offset these limitations, we propose a hybrid search graph that has sufficient resolution, and accelerates search computations by exploiting coarse transitions, when possible.

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 $\mathbf{S}_{\text{tri}}$ are the midpoints of the edges in the mesh and $\mathbf{A}_{\text{tri}}$ are the six directed transitions per triangle, two bi-directional edges for each vertex pair. This triangulation can be easily replaced by more complex solutions proposed by Mononen [2009] and Kallmann [2010], and provides a coarse discretization of the state and action space. Figure 3(a) illustrates Σ_{tri} for a simple environment. The triangulation domain Σ_{tri} provides a coarse-resolution discretization of free space in the environment, and facilitates efficient pathfinding. However, the resulting graph is too sparse to represent paths adhering to constraints such as spatial relation to an object.

To offset this limitation, we annotate objects in the environment with additional geometry to describe relative spatial relationships (e.g., `Near`, `Left`, `Between` etc.). These annotations generate additional triangles in the mesh, which expands Σ_{tri} to include states and transitions that can represent these spatial relations. Annotations, and the corresponding triangulation are illustrated in Figure 3(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 provide a dense graph representation, described below.

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 within $\frac{\sqrt{3}}{2} d_{\text{grid}}$ of the nearest point in $\mathbf{S}_{\text{tri}}$, and clamp it to that point. Each state in $\mathbf{S}_{\text{dense}}$ can have a maximum of 26 neighbors. However, in practice we have approximately 8 neighbors for planar environments. The dense domain Σ_{dense} can be precomputed or generated on the fly, depending on environment size and application requirements. However, it greatly increases the computational burden of the search due to the increased number of nodes and transitions.

4.3 Hybrid Graph

To mitigate the performance problem of Σ_{dense}, we combine Σ_{dense} and Σ_{tri} to generate a hybrid domain $\Sigma_{\text{hybrid}} = \langle \mathbf{S}_{\text{hybrid}} = \mathbf{S}_{\text{dense}}, \mathbf{A}_{\text{hybrid}} = \mathbf{A}_{\text{dense}} \cup \mathbf{A}_{\text{tri}} \rangle$. 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 add the corresponding transition $(\lambda(s), \lambda(s'))$ in $\mathbf{A}_{\text{dense}}$. The resulting hybrid domain Σ_{hybrid} has the same states as Σ_{dense} with additional transitions. These transitions are generally much longer than those in $\mathbf{A}_{\text{dense}}$, creating a low-density network of *highways* through the dense graph.

In Σ_{hybrid}, a pathfinding search can choose highways for long distances, and only use the dense graph when it is necessary. As before, the dense graph allows the planner to find paths that adhere to constraints. But when there is no strong influence of nearby constraints, the planner can take highways to improve its performance. In addition, with a planner like AD* [Likhachev et al. 2005], we can inflate the influence of the heuristic to produce suboptimal paths very quickly that favor highway selection, and iteratively improve the path quality by using dense transitions, while maintaining interactive frame rates. The performance benefits of Σ_{hybrid} are described in Section 7.1.

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 local solutions. For example, an agent who wishes to stay behind a building or outside another agent's line of sight, may choose extremely circuitous paths that satisfy these constraints. Our framework supports hard constraints which must always be met, attractors which reduce transition costs, and repellers which increase transition costs.

5.1 Hard Constraints

Hard constraints have very simple definitions comprising two fields: an object and a relative position. These two fields (explained below) allow us to define an area of influence of the constraint, where all transitions in Σ_{hybrid} that fall within the area are pruned. Hard constraints can only be `Not` constraints. In order to specify hard attractor constraints, we use a sequence of goals that the agent must navigate to (e.g., go behind the building and then to the mailbox). Hard constraints simply prevent all violating transitions

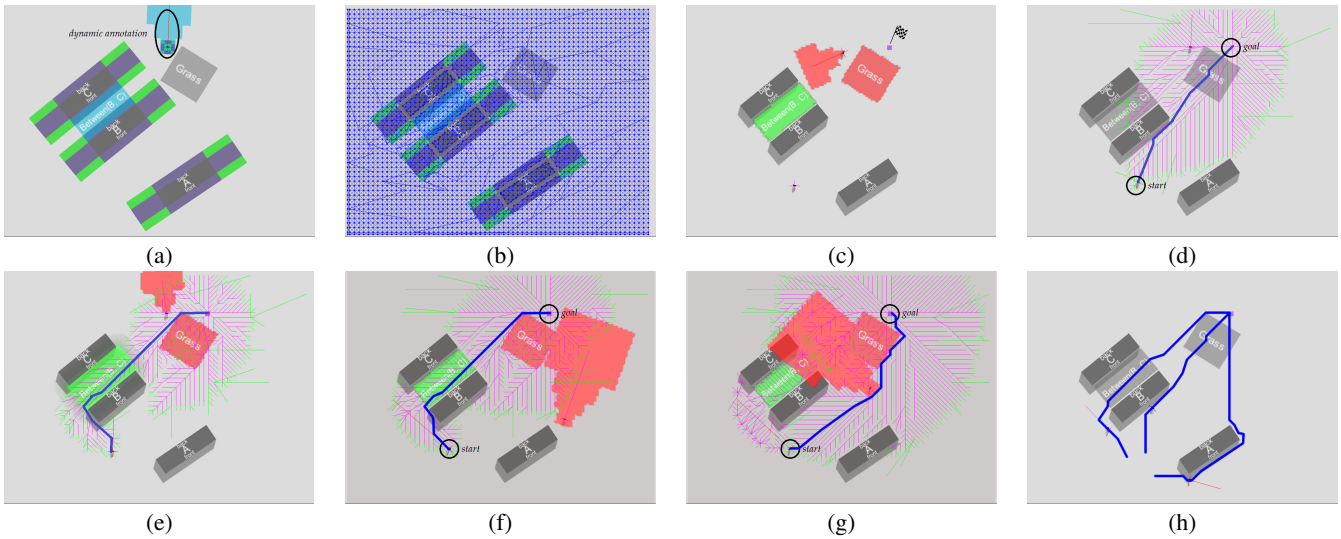


Figure 2: *(a) An environment with annotations such as* `Front`*,* `Left`*,* `Between`*, and* `LineOfSight`*. (b) Transitions in hybrid domain* Σ_{hybrid}*. (c) A specific problem instance with the following constraints:* `Not In` *Grass* $\wedge$ `Not Near LineOfSight` *(Agent)* $\wedge$ `Near Between` *(B, C). (d) Static optimal path, in absence of constraints. (e) Resulting path produced for problem instance (c). (f)–(g) Plan repair to accommodate dynamic* `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.*

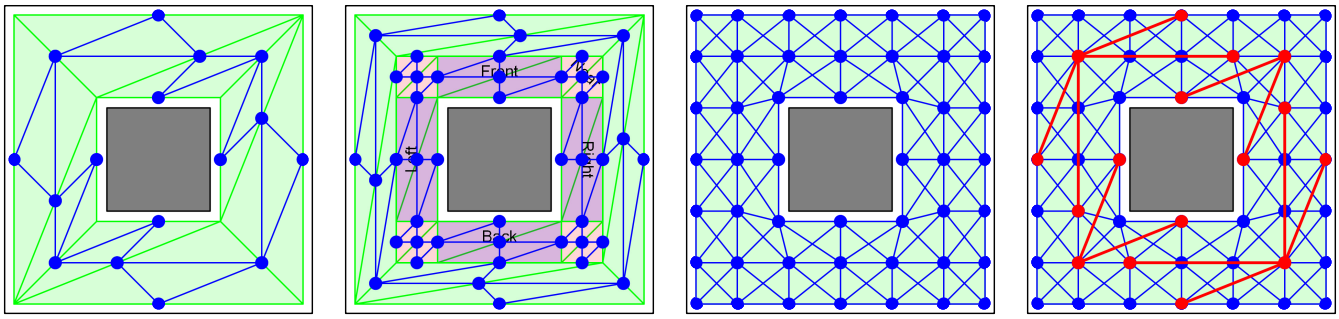


Figure 3: *(a) Environment triangulation* Σ_{tri}*. (b) Object annotations, with additional nodes added to* Σ_{tri}*, to accommodate static spatial constraints. (c) Dense uniform graph* Σ_{dense}*, for same environment. (d) A hybrid graph* Σ_{hybrid} *of (a)* Σ_{tri} *and (c)* Σ_{dense}*; highways are indicated in red.*

from being expanded during the search, thus producing a plan that always avoids the region.

5.2 Soft Constraints

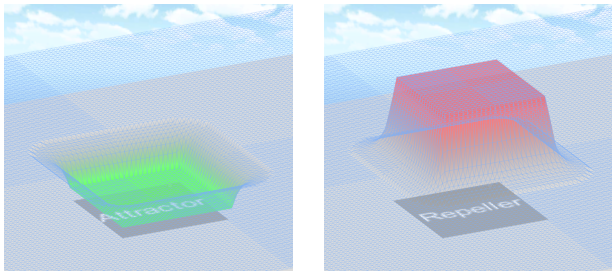
A soft constraint specification consists of three fields: (a) a preposition, (b) an object with annotations, and (c) the constraint weight. As soft constraints are an extension of hard constraints, they are simply referred to as "constraints" henceforth.

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, described below.

Annotations. An annotated object provides positions as well as semantics. Annotations define spatial regions relative to an object (or multiple objects) for the purposes of customizing prepositions. Figure 3(b) illustrates the annotations: `Back`, `Front`, `Left`, and `Right`, for a static object in the environment. Annotations can be easily added for dynamic objects as well, for example, to specify an agent's `LineOfSight`. The relationships between multiple objects can be similarly described by introducing annotations such as `Between`. The annotations define the area of influence of the constraint, relative to the position of the object.

Weight. The weight defines the influence of a constraint, and can be positive or negative. For example, one constraint may be a weak preference ($w = 1$), while another may be a very strong aversion ($w = -5$) where a negative weight indicates a repelling factor. Weights allow us to define the influence of constraints relative to one another, where one constraint may have higher priority over another, facilitating the superimposition of multiple constraints in the same area.

5.3 Multiplier Field



(a) (b)

Figure 4: *A 3D depiction of the multiplier field $m_c(\vec{x})$ for (a) attracting and (b) repelling constraints.*

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 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 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 are simply added together, and multiplied to the base 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, and hard constraints have a sharp cut-off. We model constraints as artificial potential fields and define the influence of a constraint at a particular position as a function of its radial distance from the constraint center. A decay of r^{-2} ensures that the influence is strong at small distances, and negligible beyond a certain distance. A hard constraint only affects states that are within its region of influence.
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 below). The path integral will always have the same value regardless of path sub-division.

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, due to its continuous nature, multiplier fields can be easily translated to any pathfinding system; it is not specific to graph search representations of pathfinding problems. For a single constraint c with the preposition `Near`, the cost multiplier field $m_c(\vec{x})$ is defined as follows:

$$\overline{m}_c(\vec{x}) = -w\left(k_1 + k_2 \cdot r(c,\vec{x})\right)^{-2}$$

$$m_c(\vec{x}) = \begin{cases} m_c(\vec{x}) & : \quad |\overline{m}_c(\vec{x})| < \epsilon \\ 0 & : \quad \text{otherwise} \end{cases}$$

where w is the constraint weight, $k_1 = 0.4$ and $k_2 = 0.5$ are constants affecting the sharpness and influence radius of a constraint, respectively, and $r(c,\vec{x})$ is the shortest distance between the position $\vec{x}$ and the constraint c. We define a zone beyond which the constraint has no effect and clamp its value to zero when its influence is below a certain threshold ϵ. 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. Multiplier fields for `Near` attractor and repeller are illustrated in Figure 4. For `In` constraints, $m_c(\vec{x}) = -wk_1^{-2}$ for all positions within the constraint annotation, and $m_c(\vec{x}) = 0$ for all positions outside the annotation.

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

$$m_{\mathbf{C}}(\vec{x}) = \max\left(1, m_0 + \sum_{c\in\mathbf{C}} m_c(\vec{x})\right)$$

To accommodate attractor constraints which reduce cost, we define a "base" multiplier m_0 (with a value around 3 to 10). This multiplier affects costs even in the absence of constraints, which allows attractors to reduce the cost of a transition, without it ever going below the default cost. The resulting cost multiplier never goes below 1, to preserve the optimality guarantees of the planner.

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

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

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, since the path integral is inefficient to compute on the fly, we approximate it by taking the value of the multiplier field at the midpoint of the transition,

$$M_{\mathbf{C}}(s,s') \approx m_{\mathbf{C}}\left(\frac{\vec{x}_s + \vec{x}_{s'}}{2}\right)$$

where $\vec{x}_s$ and $\vec{x}_{s'}$ are the position vectors of s and s', respectively.

6 Planning Algorithm

We use Anytime Dynamic A* [Likhachev et al. 2005] as our underlying planner, which combines the incremental planning properties of D* Lite [Koenig and Likhachev 2002] and the anytime planning properties of ARA* [Likhachev et al. 2003] to efficiently repair solutions after world changes and agent movement. It quickly generates an initial suboptimal plan, bounded by an initial inflation factor ϵ_0 which focus search efforts towards the goal. This initial plan is then improved by lowering the weight of ϵ with each new plan generated until ϵ becomes 1.0, thus guaranteeing optimality in the final solution.

AD* can interleave planning with execution by allowing the agent to move along the path, and handles start update 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, which are re-expanded 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. [2005], and describe the changes to accommodate constraint satisfaction below. Appendix A 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:

$$g(s_{\text{start}}, s) = g(s_{\text{start}}, s') + M_{\mathbf{C}}(s, s') \cdot c(s, s')$$

where $c(s, s')$ is the cost of a transition from $s \rightarrow s'$, and $M_{\mathbf{C}}(s, s')$ is the aggregate influence of all constraint multiplier fields, as described in Section 5.3. This is recursively expanded to produce:

$$g(s_{\text{start}}, s) = \sum_{(s_i, s_j) \in \Pi(s_{\text{start}}, s)} M_{\mathbf{C}}(s_i, s_j) \cdot c(s_i, s_j)$$

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 $\mathbf{C}$. States keep track of the set of constraints that influence its cost, which mitigates the need of exhaustively evaluating every constraint to compute the cost of each transition. When the area of influence of a constraint changes, the states are efficiently updated, as described below.

Accommodating Dynamic Constraints: Over time, objects associated with a constraint may change in location, affecting the constraint multiplier field which influences 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 $m_c(\vec{x})$ 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. 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 ($\mathbf{S}_c^{\text{prev}} \cup \mathbf{S}_c^{\text{next}}$) are marked as inconsistent (their costs have changed) and they must be updated. Additionally, for states $s \in \mathbf{S}_c^{\text{next}}$, if c is a hard constraint, its cost $g(s) = \infty$. Algorithm 1 provides the pseudo code for **ConstraintChangeUpdate**. The routine **UpdateState**(s), used to recompute the costs of states, is provided in Appendix A, and is modified slightly from its original definition [Likhachev et al. 2005] to incorporate the multiplier fields during cost calculation.

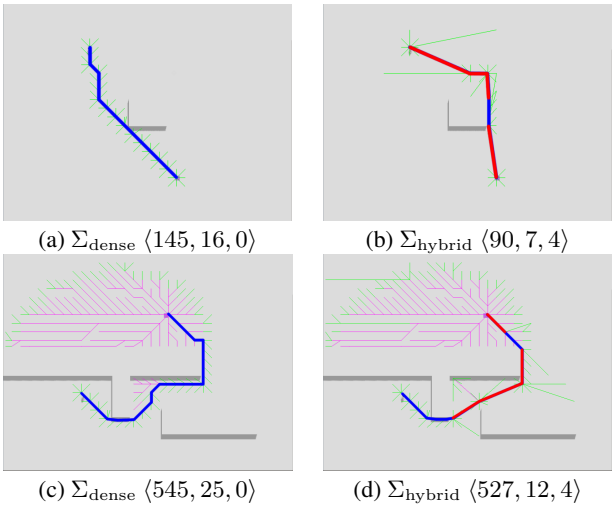
Algorithm 1 ConstraintChangeUpdate $(c, \vec{x}_{\text{prev}}, \vec{x}_{\text{next}})$

```
1:  S_c^prev = region(m_c, x_prev)
2:  S_c^next = region(m_c, x_next)
3:  for each s ∈ S_c^prev ∪ S_c^next do
4:      if pred(s) ⋂ VISITED ≠ NULL then
5:          UpdateState(s)
6:      if s' ∈ S_c^next ∧ c ∈ C_h then g(s') = ∞
7:          if s' ∈ CLOSED then
8:              for each s'' ∈ succ(s') do
9:                  if s'' ∈ VISITED then
10:                     UpdateState(s'')
```

7 Results

Our framework is implemented in C# in the Unity game engine, and uses the ADAPT platform [Shoulson et al. 2013] for character animation. Our framework is real-time, and all results described here and shown in the supplementary video were captured at 30 fps or higher.

7.1 Benefit of Hybrid Domain



(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 6: *Comparative evaluation of dense and hybrid domains. Blue indicates transitions in* $\mathbf{A}_{\text{dense}}$*, red indicates highway transitions from* $\mathbf{A}_{\text{tri}}$*. Numbers shown are* $\langle$ *number of nodes expanded, number of dense nodes chosen in path, number of highway nodes chosen in path* $\rangle$*.*

The number of nodes in the plan to reach a particular goal (effectively translating to the depth of the search) impacts the computational complexity of the search, ranging from a polynomial to an exponential effect in the worst case. The use of *highway* transitions (transitions from $\mathbf{A}_{\text{tri}}$) significantly reduce the search depth, as the length of a utilized transition from $\mathbf{A}_{\text{tri}}$ is, on average, 2 to 6 times longer than a transition in $\mathbf{A}_{\text{dense}}$. Figure 6 compares the use of Σ_{hybrid} and Σ_{dense} for the same problem instance. We observe that there is a reduction of 45 nodes expanded, for 4 highway nodes used in the plan for the problem instance in Figure 6(a),(b). The problem instance in (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.

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 a factor of $\sim 10 \cdot n_h$. 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 to where these nodes are present in the environment. This could be potentially mitigated by using high-quality navigation meshes [Mononen 2009; Kallmann 2010]. 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 which greatly accelerate and focus the search, while compromising optimality of solution.

7.2 Examples

Figure 2 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 relation-

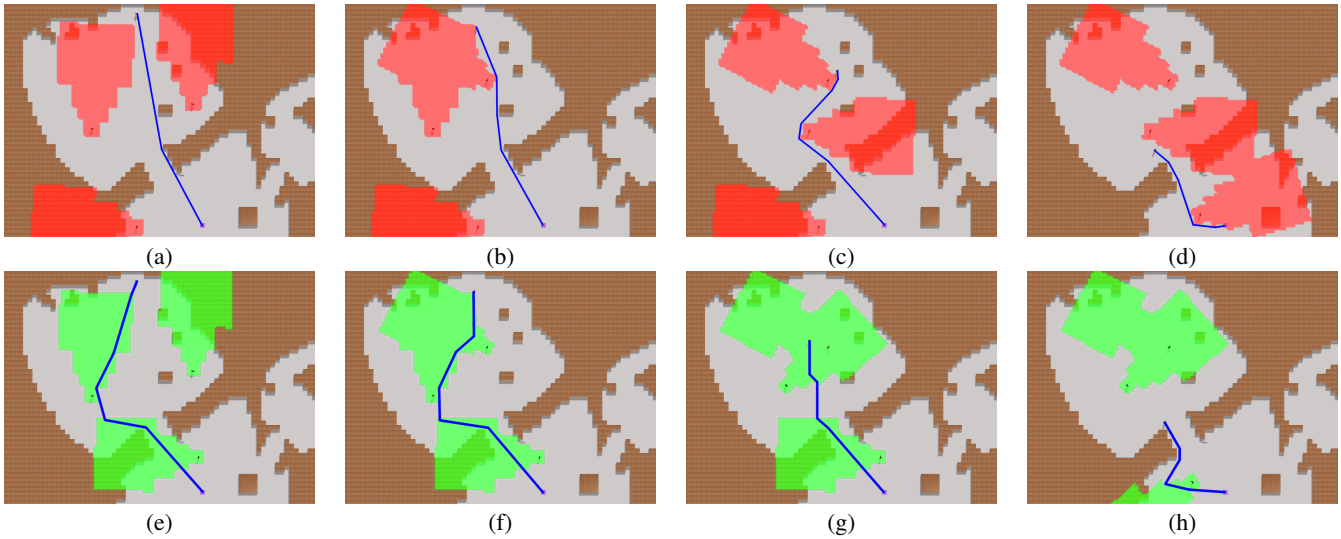


Figure 5: *(a)–(d)* `Not In LineOfSight` *constraint. (e)–(h)* `In LineOfSight` *constraint. Both cases utilize multiple dynamic agents. A user interactively moves agents and the plan is repaired to accommodate constraint change.*

ships including `Between`, `Front`, `Back`, `Left`, etc. The hybrid graph, illustrated in (b) combines the transitions in Σ_{tri} and Σ_{dense}. A specific problem instance $\mathbf{P}$, illustrated in (c), 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). Image (e) illustrates the resulting node expansion and path produced, which is drastically different from the static optimal path without any constraints, shown in (d). Images (f) and (g) illustrate 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. Image (h) illustrates multiple agents planning with different combinations of constraints.

We also demonstrate our framework on challenging game environments [Sturtevant 2012]. 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 like staying along the wall or alternating between the left and right of obstacles to produce a zig-zag path can be created by using combinations of multiple attractors and repellers, as shown in Figures 1(a)–(c). Figure 1(d) illustrates multiple agents conforming to a common set of constraints in their paths, to produce a lane formation behavior. Figures 1(e)–(h) show the use of constraints in a highway crossing scenario, where the agent avoids navigating in front of moving vehicles. Plan repair to avoid the line of sight of multiple moving agents is shown in Figure 5(a)–(d). 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 5(e)–(h). Our framework efficiently repairs the existing solution to accommodate the constraint changes. The supplementary video provides additional examples.

7.3 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 0.033s and the plan computations of multiple agents were distributed over successive frames, to ensure that the frame rate was always greater than 30Hz. 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 on average, the value of ϵ quickly converges to 1.0 to produce an optimal path, and requires a few frames to repair solutions to accommodate dynamic events. For rapid changes in the environment over many frames, the planner maybe unable to find a solution and the agent stops 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 pick the lowest possible value of m_0 to accommodate the maximum value of repelling constraints with an upper bound of 10, 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

In this paper, 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 al-

tered to stay behind buildings, walk along the 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 explore better automated solutions [Mononen 2009; Kallmann 2010] and manually annotated waypoint graphs to improve computational performance. The annotations for an object are currently limited to a single object (e.g., `Front`) or for object-pairs (e.g., `Between`). Static analysis of the environment could potentially yield automatic annotation generation for more complex spatial relationships, and is the subject of future exploration. We have only considered spatial constraints in this paper, but our framework is general and extensible to other problem domains. For future work, we would like to extend our specification to enforce movement as well as temporal constraints.

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 prudent 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 the subject of future exploration.

A Anytime Dynamic Planner

EventHandler (Algorithm 2 [24–31]) monitors events in the simulation and triggers appropriate routines. **ComputeOrImprovePath** (Algorithm 2 [15–23]) is invoked each 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 [Likhachev et al. 2005]. 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 [Likhachev et al. 2005] 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 [Koenig and Likhachev 2002] 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 ($g(s) \neq rhs(s)$) which 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 towards 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 already expanded nodes that become inconsistent due to 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 due to 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 since 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 cost and their new cost. 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- and over-consistent states, as shown in Algorithm 2 [1–5]. 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 below, while 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 towards 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. Alg 2 [32–40] provides the routine to handle start movement.

GoalChangeUpdate. Alg 2 [41–44] clears plan data and resets ϵ whenever the goal changes and plans from scratch at the next step.

References

AL MARZOUQI, M., AND JARVIS, R. 2011. Robotic covert path planning: A survey. In *Robotics, Automation and Mechatronics (RAM), 2011 IEEE Conference on*, 77–82.

ANDRÉ, E., BOSCH, G., HERZOG, G., AND RIST, T. 1986. Characterizing trajectories of moving objects using natural language path descriptions. In *In: Proc. of the 7th ECAI*, 1–8.

ARKIN, R. 1987. Motor schema based navigation for a mobile robot: An approach to programming by behavior. In *Robotics and Automation. Proceedings. 1987 IEEE International Conference on*, vol. 4, 264 – 271.

BHATTACHARYA, S., LIKHACHEV, M., AND KUMAR, V. 2012. Topological constraints in search-based robot path planning. *Auton. Robots 33*, 3, 273–290.

BHATTACHARYA, S., LIKHACHEV, M., AND KUMAR, V. 2012. Search-based path planning with homotopy class constraints in 3d. In *AAAI*.

Algorithm 2 Anytime Dynamic Planner

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: CLOSED = CLOSED $\cup\, s$
21: **else**
22: $g(s) = \infty$
23: **UpdateState**(s)

24: **function EVENTHANDLER**
25: **if** START_CHANGED **then**
26: **StartChangeUpdate** (s_c)
27: **if** GOAL_CHANGED **then**
28: **GoalChangeUpdate** (s_{new})
29: **if** CONSTRAINT_CHANGED **then**
30: **for each** constraint change c **do**
31: **ConstraintChangeUpdate** $(c, \vec{x}_{\mathrm{prev}}, \vec{x}_{\mathrm{next}})$

32: **function STARTCHANGEUPDATE**(s_c)
33: **if** $s_c \notin \Pi(s_{\mathrm{start}}, s_{\mathrm{goal}})$ **then**
34: **ClearPlanData**()
35: $\epsilon = \epsilon_0$
36: **else**
37: $s_{\mathrm{start}} = s_c$
38: **for each** $s \in$ OPEN **do**
39: Update **key**(s)
40:

41: **function GOALCHANGEUPDATE**(s_{new})
42: **ClearPlanData**()
43: $\epsilon = \epsilon_0$
44: $s_{\mathrm{goal}} = s_{new}$

DECHTER, R., AND PEARL, J. 1985. Generalized best-first search strategies and the optimality af a*. *J. ACM 32*, 3, 505–536.

GERAERTS, R. 2010. Planning short paths with clearance using explicit corridors. In *ICRA*, IEEE, 1997–2004.

GOLDENSTEIN, S., KARAVELAS, M., METAXAS, D., GUIBAS, L., AARON, E., AND GOSWAMI, A., 2001. Scalable nonlinear dynamical systems for agent steering and crowd simulation.

HART, P., NILSSON, N., AND RAPHAEL, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on 4*, 2, 100–107.

HART, P. E., NILSSON, N. J., AND RAPHAEL, B. 1972. Correction to "a formal basis for the heuristic determination of minimum cost paths". *SIGART Bull.*, 37, 28–29.

HELBING, D., AND MOLNÁR, P. 1995. Social force model for pedestrian dynamics. *Phys. Rev. E 51*, 5 (May), 4282–4286.

HERNANDEZ, E., CARRERAS, M., GALCERAN, E., AND RIDAO, P. 2011. Path planning with homotopy class constraints on bathymetric maps. In *OCEANS - Europe*.

KALLMANN, M. 2010. Shortest paths with arbitrary clearance from navigation meshes. In *ACM SIGGRAPH/Eurographics SCA*, 159–168.

KAPADIA, M., AND BADLER, N. I. 2013. Navigation and steering for autonomous virtual humans. *Wiley Interdisciplinary Reviews: Cognitive Science*, n/a–n/a.

KAPADIA, M., SINGH, S., HEWLETT, W., AND FALOUTSOS, P. 2009. Egocentric affordance fields in pedestrian steering. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, ACM, New York, NY, USA, I3D '09, 215–223.

KAPADIA, M., SINGH, S., HEWLETT, W., REINMAN, G., AND FALOUTSOS, P. 2012. Parallelized egocentric fields for autonomous navigation. *The Visual Computer*, 1–19. 10.1007/s00371-011-0669-5.

KAPADIA, M., BEACCO, A., GARCIA, F., REDDY, V., PELECHANO, N., AND BADLER, N. I. 2013. Multi-domain real-time planning in dynamic environments. In *Proceedings of the 12th ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, ACM, New York, NY, USA, SCA '13, 115–124.

KOENIG, S., AND LIKHACHEV, M. 2002. D* Lite. In *National Conf. on AI*, AAAI, 476–483.

LIKHACHEV, M., GORDON, G. J., AND THRUN, S. 2003. ARA*: Anytime A* with Provable Bounds on Sub-Optimality. In *NIPS*.

LIKHACHEV, M., FERGUSON, D. I., GORDON, G. J., STENTZ, A., AND THRUN, S. 2005. Anytime Dynamic A*: An Anytime, Replanning Algorithm. In *ICAPS*, 262–271.

MONONEN, M., 2009. Recast: Navigation-mesh construction toolset for games. http://code.google.com/p/recastnavigation/.

PARIS, S., PETTR, J., AND DONIKIAN, S. 2007. Pedestrian reactive navigation for crowd simulation: a predictive approach. *Comput. Graph. Forum 26*, 3, 665–674.

PELECHANO, N., ALLBECK, J. M., AND BADLER, N. I. 2007. Controlling individual agents in high-density crowd simulation. In *Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation*, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, SCA '07, 99–108.

PELECHANO, N., ALLBECK, J. M., AND BADLER, N. I. 2008. *Virtual Crowds: Methods, Simulation, and Control*. Synthesis Lectures on Computer Graphics and Animation. Morgan & Claypool Publishers.

PHILLIPS, M., HWANG, V., CHITTA, S., AND LIKHACHEV, M. 2013. Learning to plan for constrained manipulation from demonstrations.

REYNOLDS, C. 1999. Steering Behaviors for Autonomous Characters. In *Game Developers Conference 1999*.

SCHUERMAN, M., SINGH, S., KAPADIA, M., AND FALOUTSOS, P. 2010. Situation agents: agent-based externalized steering logic. *Comput. Animat. Virtual Worlds 21* (May), 267–276.

SHIMODA, S., KURODA, Y., AND IAGNEMMA, K. 2005. Potential field navigation of high speed unmanned ground vehicles on uneven terrain. *Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on*, 2828–2833.

SHOULSON, A., MARSHAK, N., KAPADIA, M., AND BADLER, N. I. 2013. Adapt: the agent development and prototyping testbed. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ACM, New York, NY, USA, I3D '13, 9–18.

SINGH, S., KAPADIA, M., HEWLETT, B., REINMAN, G., AND FALOUTSOS, P. 2011. A modular framework for adaptive agent-based steering. In *Symposium on Interactive 3D Graphics and Games*, ACM, New York, NY, USA, I3D '11, 141–150 PAGE@9.

SINGH, S., KAPADIA, M., REINMAN, G., AND FALOUTSOS, P. 2011. Footstep navigation for dynamic crowds. *Computer Animation and Virtual Worlds 22*, 2-3, 151–158.

STURTEVANT, N. 2012. Benchmarks for grid-based pathfinding. *Transactions on Computational Intelligence and AI in Games 4*, 2, 144 – 148.

TREUILLE, A., COOPER, S., AND POPOVIĆ, Z. 2006. Continuum crowds. *ACM Trans. Graph. 25*, 3, 1160–1168.

VAN DEN BERG, J., LIN, M. C., AND MANOCHA, D. 2008. Reciprocal velocity obstacles for real-time multi-agent navigation. In *IEEE International Conference on Robotics and Automation*, IEEE, 1928–1935.

WARREN, C. 1989. Global path planning using artificial potential fields. In *Proceedings of IEEE ICRA*, vol. 1, 316–321.

WARREN, C. 1990. Multiple robot path coordination using artificial potential fields. In *Proceedings of IEEE ICRA*, vol. 1, 500–505.

XU, Y. D., AND BADLER, N. 2000. Algorithms for generating motion trajectories described by prepositions. In *Computer Animation 2000. Proceedings*, 30–35.