Posthoc — The Visualisation Platform for Search

Kevin Zheng, Daniel Harabor, Michael Wybrow

Monash University
kevin.zheng@monash.edu, daniel.harabor@monash.edu, michael.wybrow@monash.edu

Abstract

Search, especially pathfinding search, is a foundational problem-solving technique in Computer Science for sequential decision-making problems. Such algorithms appear widely in the academic literature and they have found broad applicability including in personal navigation, robotics and computer games. Despite their importance, search algorithms can be challenging for practitioners to implement and difficult for learners to understand. In this work, we present Posthoc, a visualisation and debugging tool which aims to improve the situation. Our approach relies on *search traces*, textual records of key operations that occur during the search process; e.g., node expansion, successor generation and other events of interest. We employ search traces to visualise the decision-making process and to construct domain-specific representations for each event. We show how these traces can be used—in a variety of contexts—to inspect, debug, and better understand search algorithms. Finally, we demonstrate Posthoc in a range of different real-world case studies.

Introduction

Search—informed traversal of a state space—describes an important class of problem solving approaches from the area of Planning and Scheduling. This topic finds relevance in many industrial settings; e.g., AI (Wilkins 2014), Game Development (Rabin 2019), Robotics (Kavraki and LaValle 2016), Routing (Bast et al. 2016) and more.

When solving search problems researchers and practitioners need to constantly translate between the search-space representation of an algorithm and the state-space representation of the problem. Often, this is done by analysing textual descriptions of problem solving processes and generated solutions. This is a cognitively demanding task that is tedious and also error-prone. Visualisations are invaluable tools for reducing cognitive load and aiding understanding. Researchers and practitioners working on search have thus developed a variety of visual tools: for debugging, testing and verification of algorithmic correctness, as well as demonstrators that showcase solutions and program execution to others. In the same vein, educators often create visualisations to help students understand problem solving ideas and to compare algorithmic approaches.

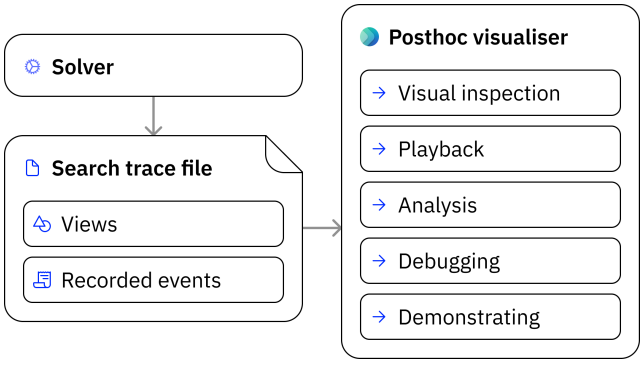

Figure 1: Basic usage of the Posthoc framework. The search trace file is a visualisable record of solver-generated events. These can be displayed in the Posthoc visualiser.

Unfortunately it is often challenging to incorporate visualisations into development, demonstration and teaching workflows. Building such tools from scratch is time-consuming and in situations where related tools do exist, they often fall short: in terms of what information they can present, how they can present it, how readily they can be integrated with specific solvers and how effectively they can be extended. These issues arise because of gaps, between intended use-cases for which existing tools are designed, and the context-specific requirements of their users, who often want to solve and visualise problems from a new or unsupported domain or ask one-off questions regarding program execution and intermediate state.

In this paper, we present Posthoc, a new framework that lowers the barrier to entry for the development of custom visualisations that support sequential decision-making programs. Our first contribution is the *search trace format*, a drop-in replacement for output logs. Search traces contain structured descriptions of solver-specific operations and events (e.g., node expansions). These outputs are further enhanced using a *view definition language*, which tells how to translate logged information into simple graphics. Our second contribution is the Posthoc application[1], a web environment where search traces can be loaded for playback, inter-

[1] Available at https://posthoc.pathfinding.ai, source available at https://github.com/shortestpathlab/posthoc-app

Listing 1: Example search trace from a grid-based pathfinding problem. We show a source and node expansion event which are visualised as square cells. The `x` and `y` attributes on line 18, 19 and 28, 29 drive the position of cells via a `${{}}` expression (line 10, 11). The example shows additional values related to search events. These can be integrated to enhance information presented in the visualisation, e.g., event type, cost labels (f, g, h), and back pointers (pId). See full trace in Posthoc's documentation.

```
1  version: 1.4.0
2  # View definition
3  views:
4    main:
5      - $: rect
6        alpha: 1
7        fill: ${{ color[$.type] }}
8        height: 1
9        width: 1
10       x: ${{ $.x }}
11       y: ${{ $.y }}
12 # One or more events
13 events:
14   - id: DHQPrdoA
15     pId: null
16     type: source
17     # Visualisation parameters
18     x: 277
19     y: 186
20   - id: DHQPrdoA
21     pId: null
22     type: expanding
23     # Search progress
24     f: 147.05441
25     g: 0
26     h: 147.05441
27     # Visualisation parameters
28     x: 277
29     y: 186
```

rogation and display. Figure 1 illustrates the basic workflow. Our application has a shallow learning curve: basic chronological information from a solver (e.g., parent-successor relations) is sufficient for playback, debugger-style interrogation (including breakpoints) and visualisation of decision-trees. Further enhancements, to produce domain-specific visualisations (e.g., maps, networks, other layouts), requires only instantiation of simple drawing primitives. Visualisations are therefore easily generated, modified, disseminated and reproduced. Being a web-based tool means Posthoc is platform-independent and does not require any setup or installation. We give a description of the tool, its design and operation, and we explore its efficacy through a variety of real-world use-cases.

Related Work

Existing visualisation support tools tend to fall in four categories outlined below. Each suffers from drawbacks of being

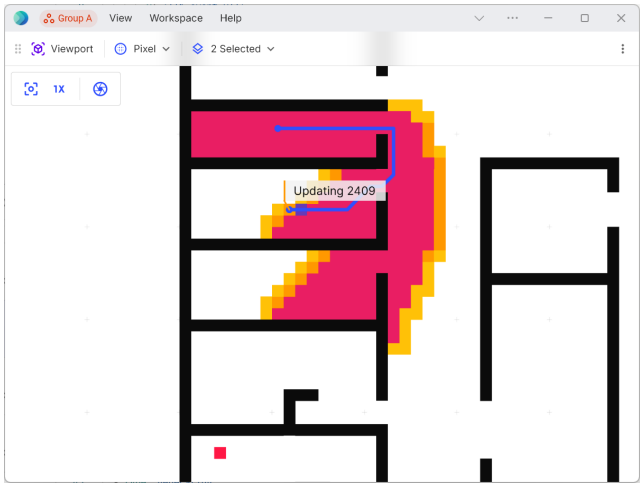

Figure 2: The trace in Listing 1 visualised in Posthoc.

either (1) narrowly focused, (2) tightly coupled with a backend/solver, or (3) having limited reusability.

Bespoke visualisations are one-shot demonstrators that illustrate a single algorithm or idea. For example, Red Blob Games (Patel 2020) features many such visualisations that help students understand key concepts (e.g., terrain generation) from the discipline of game development. These visualisations are custom-built programs with little room for adaptation or reuse.

Domain-specific visualisations are demonstrators and simulators that compare several approaches on a specific algorithmic challenge (Xu 2020; Sturtevant 2020). Recent examples of such programs include MAES (Andreasen et al. 2022), a visualisation and debugging environment for robotics applications, and PDSim (De Pellegrin and Petrick 2024) a solution simulator for PDDL. These utilise a 3D game engine (Unity 3D). Though more flexible than bespoke visualisations, they are nevertheless domain-specific and tend to be tightly integrated with a particular solver.

Active learning systems describe programs that provide step-by-step guidance and immediate feedback with the express purpose to teach a certain family of related problem-solving techniques (Sánchez-Torrubia, Torres-Blanc, and Lopez-Martinez 2009; Borissova and Mustakerov 2015). As an example, Algorithm Visualizer (Park 2020) lets students explore problem-solving techniques, e.g., dynamic programming, divide-and-conquer, branch-and-bound. One can generate and solve simple problem instances, modify algorithmic parameters, and compare the execution processes and results, albeit only through simple tree, graph, list-based visualisations. Additionally, algorithms must be written in one of the few languages supported by its API.

Search profilers are analysis and profiling software. Each profiler has a handful of approaches for visually presenting this information to practitioners to communicate how effectively a particular algorithm solves a given problem. Historically, this is done via plots, charts, and trees (Vallati and Kitchin 2020), though some profilers feature more creative approaches, like the *dovetail* for conveying action depen-

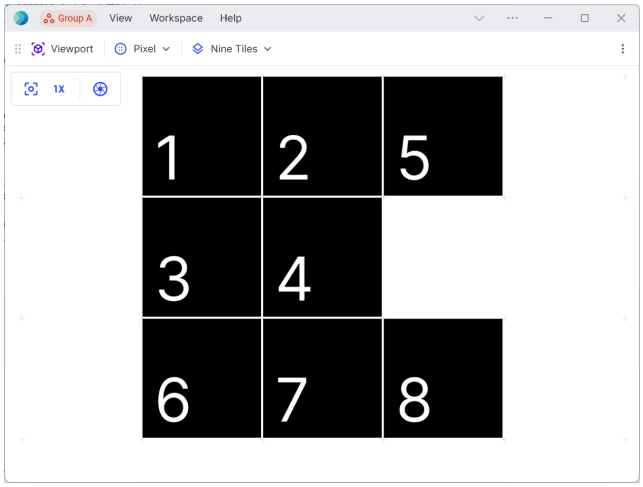

Figure 3: The trace in Listing 2 visualised in Posthoc.

dencies and sequencing (Magnaguagno et al. 2017). In the same vein, search profilers also exist for constraint programming (Simonis et al. 2010; Bauer et al. 2010; Shishmarev et al. 2016), typically presenting procedures as search trees. Most profilers are unconcerned with domain visualisation—if users wish to see partial solutions in context of the domain, they will still need to pick up other visualisation tools, or manually create them.

Chronologies

Posthoc operates on top of an abstraction for methodologies that solve planning problems. In this abstraction, we say that programs iterate over a series of program states (*events*). Although execution is sequential, the logic of decision-making processes is branching. We can better represent the relationship of possible plans as a *chronology*—a sort of directed tree or graph. For example, successor events represent an improvement or subsequent step over a predecessor.

Each chronological event is associated with a particular state. Usually, states have a visual representation, i.e., a configuration of the system in a particular domain. These visual representations should help researchers and learners "see what the algorithm is thinking thus far".

Recording Chronologies — *Search Trace*

Posthoc builds on this idea by creating a standardised format for describing chronologically related events. It takes as input a *YAML* event log, which we call a *search trace* or simply *trace*[2]. It is a highly generic textual file format that is agnostic of any solver or visualiser; making it easily producible, transportable, and shareable. Usually, it contains two sections: a list of recorded `events` from the solver, and a list of `views` defining custom visualisations in the *view definition language*. Search programs are often instrumented to produce such logs, during algorithmic development, and some solvers output event logs as part of their core functionality; e.g., Warthog (Harabor 2024), a library for pathfinding search. One can use the *search trace* format as a drop-in replacement, or additional, logging format for these solvers. For simple visualisations, one can also write these by hand.

[2]Full documentation of the Search Trace format is available at https://posthoc.pathfinding.ai/docs/search-trace

Listing 2: This example shows the *view definition language* code to render a sliding tile puzzle. This code transforms arbitrary events (descriptions of the board) into visualisations. `views.tile` (line 9) draws a single tile in the puzzle. It is represented as one `rect` primitive (line 11-17), and depends on the `$.row` and `$.col` variables to set the x and y positions as well as to retrieve the label from the board. `views.main` is the entry point for the view definition program. The `row` and `col` variables are computed in the `main` view, and consumed in the `tile` view to render each board tile. Some lines are omitted for concision; see full trace in Posthoc's documentation.

```
1  version: 1.4.0
2  views:
3    main:
4    - $: tile
5      $for:
6        $to: ${{ $.width * $.height }}
7      col: ${{ Math.floor($.i / $.
           width) }}
8      row: ${{ $.i % $.width }}
9    tile:
10   - $: rect
11     clear: true
12     fill: ${{ theme.foreground }}
13     height: 0.98
14     label: ${{ $.board[$.col][$.row]
           }}
15     width: 0.98
16     x: ${{ $.row }}
17     y: ${{ $.col }}
18 events:
19 - board:
20   - [1, 2, 5]
21   - [3, 4, 8]
22   - [7, x, 9]
23   height: 3
24   width: 3
```

Events

The `events` array is a list of records, each of which describes the internal state of the program at a specific point in time. Events may contain a few basic labels. The presence of the identifier `id` label marks events as belonging to *nodes* (that is, specific problem configurations); the same node may show up multiple times as their state changes. Furthermore, the presence of the parent identifier label, `pId`, classifies the events belonging to a *chronology*, and hence visualisable as a tree or graph. The `type` property is a label describing the state of the program. To "see" what an event looks like in context, we allow the user to instantiate various drawing

primitives with the *view definition language*.

Listing 1 shows an example of a search trace representing a search procedure, in this case for a grid-based pathfinding problem solved via A* (Hart, Nilsson, and Raphael 1968) (visualised in Figure 2). This is output from a solver which communicates its algorithmic process in a *language of search* (other solvers may use entirely different terminology). In this language *operations* (recorded as events) are performed on *nodes* (i.e., intermediate states) in order to simulate an *agent* performing actions, usually with the objective being transitioning from a *source* to a *target* node. There are two universal node operations found in all search algorithms.

- **Expand** occurs when the search processes an existing node, typically identifying all actions extending from this state.
- **Generate** produces a new node representing the commitment to an action. Each such node is referred to as a *successor*.

Search algorithms generally extend this minimal set of operations. For graph-based search, nodes may be *relaxed* and *closed* (Hart, Nilsson, and Raphael 1968). We may also consider the act of identifying *start* and *target* nodes as operations. In the trace, the `type` property holds the name of the current operation. The `id` label uniquely specifies the node, and the `pId` label identifies its predecessor, usually the node that *generated* it.

Search usually proceeds according to some strategy, often informed by a *heuristic*. We record this information in the trace. Common labels include the `g-value`, denoted $g(n)$, telling the cost of the current solution, from the start node to n. The `h-value` label, denoted $h(n)$, estimates the cost from n to the target. Finally, the `f-value` label, defined as $f(n) = g(n) + h(n)$, bounds the cost of a solution from start to target via n. Moreover, solutions may be required to meet some quality criteria (in terms of optimality or total cost) and it must satisfy constraints placed upon the agent. For example in Listing 1, labels x and y store the coordinates of the agent's current location, and are used as visualisation parameters referenced in the view definition (Listing 1, lines 10 and 11). In this case, we can verify that the agent does not, for example, collide with terrain.

Views

To support the variety of visualisations that practitioners want to create, the `views` array allows users to quickly write custom visualisations in a declarative *view definition language*. It strikes a balance between ease of use, expressiveness, and extensibility; in our case studies, it proves to be a suitable medium for quickly defining graphics to visualise a wide variety of problems.

View Composition and Primitives Visualisations are composed of views—reusable assets made up of one or more child views or *primitives*. They define how a visualiser should translate each event in the `events` list into graphics.

In this system, primitives are views defined by the renderer. For a 2D renderer, this may be `rect`, `circle`, etc. The expressiveness of this language originates from the following—views hold properties that inherit values from their parent or which are read from the current event. The system also defines several basic operators as properties, such as `$if` and `$for`. Additionally, one can write expressions as part of any property. These are thoroughly documented in our user guide.

During preprocessing, each event is applied to the views defined in the trace, and the views are flattened into a list of primitives. Then, the primitives are passed to the renderer to be displayed.

Expressions Expressions open the door to more complex and dynamic visualisations. The *view definition language* introduces computed properties for performing mathematical operations and data transformations. These are properties surrounded by double curly braces `${{ }}`, whose contents will be evaluated by an interpreter. Listing 1 illustrates this mechanism on lines 7, 10, and 11. The `$` symbol indicates a reference to properties in current scope, falling back to properties from the current event. In the Posthoc visualiser, these are evaluated via a JavaScript virtual machine. It also provides several global symbols, e.g. `color` (Listing 1, line 7), for styling convenience.

Listing 2 shows a search algorithm applied to a sliding tile puzzle. This trace is created with the *Piglet* (Chen et al. 2024) solver. In this case, the interpretation of *nodes* is the puzzle itself with the moving pieces being in a particular arrangement. Notice the board state being recorded as part of each event (Listing 2, lines 19–24), as well as the *view definition language* expressions and constructs describing how to iterate over the board state (Listing 2, lines 5–8) and translate events to visual elements (Listing 2, lines 9–17). We show this problem visualised in Figure 3. Notice also how the trace, including board state information in the event list, remains human readable.

Posthoc Visualiser

The Posthoc visualiser is a web-based environment for visualising and interacting with search traces. Considered the primary artifact of the Posthoc project, it allows users to easily import search traces, play back and inspect search or execution processes, perform debugging and analysis tasks, and share visualisations. Posthoc has a shallow learning curve, offering copious examples, documentation, and community support. As a web-based tool, it is platform-independent, zero-setup, and does not require installation or compilation; we offer additional standalone Linux, MacOS, and Windows builds.

Figure 4 demonstrates a typical Posthoc visualiser session, in this case of a user interacting with bidirectional A* search (Ikeda et al. 1994) on a large network map. The user has access to a list of imported files (Figure 4a), events (Figure 4c), playback controls (Figure 4b), and a list of breakpoints (Figure 4e); they have open both the viewport (Figure 4d) and graph (Figure 4g) views. They have clicked on an element in the viewport to inspect it (Figure 4f).

Inspection and Playback

Posthoc allows users to explore recorded data via simple playback mechanisms (Figure 4b). Events are parsed and

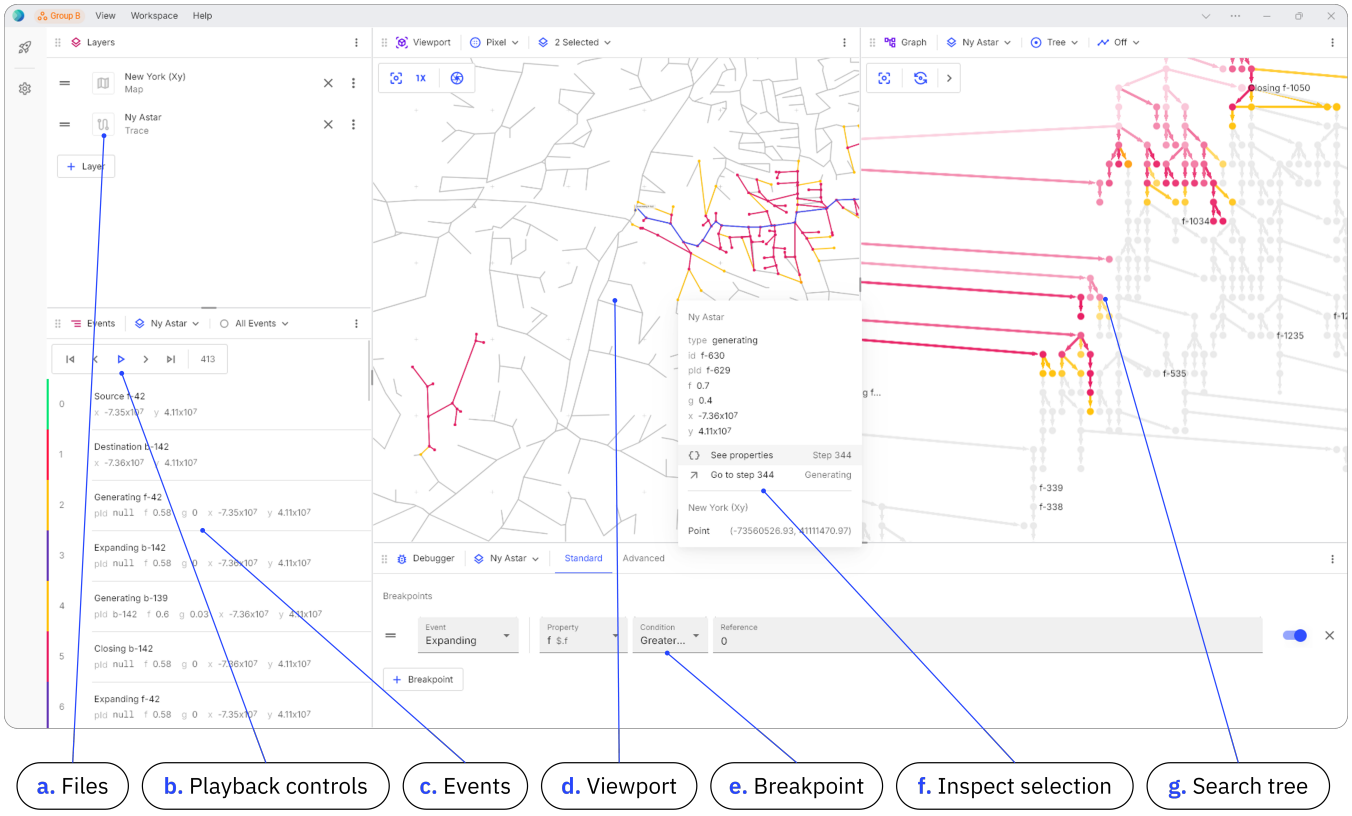

Figure 4: Posthoc visualiser user interface, labelled. The problem instance shown is a bidirectional A* search (Ikeda et al. 1994) on a New York network map. This is produced via Warthog, a pathfinding search solver.

visualised in input order, which allows the user to inspect the process: step by step, to verify the correctness of each operation, or holistically, to acquire general insights into the search process (e.g., where were the "hard bits").

The system offers two views: a *chronology* view (Figure 4g, labelled *Graph*) and a domain-specific *viewport* (Figure 4d, labelled *Viewport*). In each view the user can select elements to better understand the search process (Figure 4f).

The *graph* view visualises chronologies as a tree or directed graph, offering automatic layout of the recorded decision events; it is also built to render millions of nodes. The graph view is automatically enabled for search traces where events contain `id` and `pId` properties. It also allows users to visualise the changes of numeric properties across events.

The viewport displays visualisations written in the *view definition language*. Sometimes, search traces should be accompanied by an environment (maps) to be understood. Posthoc has built-in support for importing common map files from the pathfinding search community: grid (*.map*), navigation mesh (*.mesh* and *.poly*) and 2D network (*.xy*) maps out-of-the-box (Figure 4d).

Side-by-side Inspection Commonly, the user wishes to correlate the execution of the search with the domain representation of each event. The *graph* and *viewport* panels could be laid side-by-side to inspect both the search process and partial plan 4. In this case, playing back the search trace would update both views.

Comparison When multiple traces are loaded, Posthoc facilitates comparisons between different solver outputs, either superimposed or side-by-side, useful when researchers want to compare their work with a known good baseline.

Debugging

One of the most time-consuming aspects of implementing a search algorithm is the detection of runtime bugs which cause infeasible or incorrect solutions or lead to incorrect proofs; e.g., an unfounded optimality "guarantee". Detecting such bugs requires a detailed understanding of the algorithmic state and how it changes during search; currently this is done via log inspections or assertion statements.

In Posthoc, *breakpoints* are conditional statements about events. Posthoc allows users to set two types of breakpoints. Standard breakpoints can be set via a list-based user interface (Figure 4e). For search, one would usually set them on `id`, `parent`, `f-value`, `g-value` or `h-value` (Hart, Nilsson, and Raphael 1968). Breakpoint rules compare the label value against a user-specified value using standard comparison operators. Users can also use JavaScript expressions to pause on custom conditions.

During playback of the trace, if the current event satisfies one of the breakpoint conditions we pause the playback and show a message which gives the user an opportunity to in-

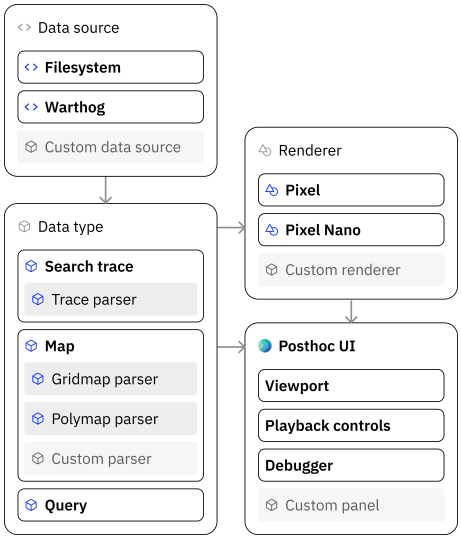

Figure 5: A simplified representation of Posthoc's architecture. The boxes labelled *data source*, *data type*, and *renderer* are abstractions. Posthoc defines a set of built-in implementations for those abstractions. The grey boxes labelled "Custom" indicate a few areas where third-parties can enable additional features through our extensions API.

spect the state of the algorithm at a point of interest. Events that satisfy breakpoint conditions are given a red dot. The playback can be subsequently resumed and execution of the trace continues until the next breakpoint or until the last event is processed.

Search Queries On-the-Fly

In pedagogical settings, it is often desirable to interact directly with a solver program and observe its output. Search queries help users to better understand search algorithms by observing how they tackle different problems, in real time. Integrating a solver with Posthoc is straightforward: the user specifies which executable to invoke and which input problem file. Solvers return output as *traces*. A more complex use case allows the user to specify parameters directly from the visualiser. In this case, the integration with the solver is required to map input from the visualiser to the underlying problem format expected by the solver, so that new problem instances can be created on-the-fly.

Sharing Visualisations

It is a common use case for researchers to share their work for purposes of discussion, collaboration, dissemination etc. Search trace files are easily shared, due to their textual nature. Full visualisations, potentially consisting of multiple search trace files, map files, and layer configuration, can be saved as a *workspace* file. This is a Posthoc-specific, compressed, binary format that lets users easily share their work from within Posthoc.

System Architecture

To ensure Posthoc remains a stable and modular platform for extensions and community contributions, we introduce several abstractions regarding how data can be imported, processed, rendered, and interacted with. With reference to Figure 5, a *data source* represents a repository for importable or visualisable content, such as the file system. Next, a *data handler* is defined for each *data type*. These should (1) provide some sort of user interface for selecting and importing data from a data source, and (2) process imported data into a structure understood by the *renderers* and other Posthoc subsystems, like the *debugger*, *event list*, and *playback controls*. The arrows indicate an interface or protocol to facilitate data flow between systems. The majority of Posthoc's functionality, for example the mechanism for processing search traces, are built as implementations of these abstractions; we intend for the community to do the same. Figure 5 also gives an idea what sort of features can be added via our extensions API.

Posthoc is designed to maintain responsiveness when used with realistically-sized problem instances and traces—Figure 4 shows Posthoc rendering a graph representing New York's road network, consisting of 264,346 nodes and 730,100 edges. Since search algorithms can output traces with millions of events and sizes in the hundreds of megabytes, we implement a variety of optimisations in loading, processing, and rendering content, including the use of cutting-edge web technology, such as virtualisation, WebWorkers, OffscreenCanvas, WebAssembly and WebGL2. We also ensure that Posthoc is ergonomic enough to fit snugly into researchers' workflows. Posthoc's panel-based user interface is intuitive and highly configurable (Figure 4). We offer conveniences like drag-and-drop, touch gesture support, and a comprehensive library of examples and documentation.

Performant Rendering

Posthoc offers two renderers, *Pixel* and *Pixel Nano*. *Pixel* is a powerful, high-performance, asynchronous, multi-process, tile-based WebGL2 renderer. *Pixel* indexes scene elements in an R-Tree (Guttman 1984) and renders on a separate thread. It can efficiently render the large-scale pathfinding search problems that appear in many applications. For example, it can easily handle searches that involve tens of thousands of nodes, thousands of agents, and maps of the scale of major cities (Figure 4d). Furthermore, it also supports infinite zoom. *Pixel Nano* is a simpler fallback renderer that runs on the main thread for cases where, for example, the browser does not support APIs required for *Pixel* to run.

Usage Examples

To demonstrate the efficacy of the Posthoc framework, we showcase several real-world and synthetic cases from the area of pathfinding, a popular application of search-based solvers. For more examples of other algorithms and visualisations of different domains, see the Posthoc website.

Visualising Path Planning Algorithms

Path planning, or pathfinding, is a problem archetype soliciting a sequence of actions to bring an agent from a source to a goal location. It is a good example for where partial plans have a visual embedding, typically a Euclidean environment. Pathfinding problems appear in a wide variety of different domains and they are solved with algorithmic approaches that are just as diverse.

Path Planning on a Grid Grid-based pathfinding is a popular topic which receives substantial attention from practitioners in computer game development (Botea et al. 2013). In this domain the operating environment is discretised into tiles and the agent is allowed to move from one adjacent tile to the next.

As an example, we use an instrumented version of the Warthog pathfinding library which supports a variety of grid-based pathfinding techniques. Listing 1 exemplifies a search trace for grid-based path planning, Figure 2 shows it visualised. Each event records the x and y coordinates for each tile; tiles are appropriately visualised as a coloured square. See the Posthoc website for more examples of grid-based path planning visualised, including Jump Point Search (Harabor and Grastien 2014), a leading grid-based method whose implementation is found in Warthog, as well as a heat-map version of an A* search.

Path Planning on a Navigation Mesh A navigation mesh is a pathfinding data structure that divides the operating environment into convex polygons. When moving on a mesh, the agent is free to enter and exit each polygon at any angle. Compared to path planning on a grid in which agents can only move from one fixed point to another, the search space is continuous. Polyanya (Cui, Harabor, and Grastien 2017) is a recent and state-of-the-art algorithm for solving such problems. Although Polyanya instantiates the A* algorithm, it uses a specialised and domain-specific representation where each node is a tuple (I, r) where r is the last turning point on a path from the start location and $I = [a, b]$ is a contiguous interval of points from a polygon edge such that every point $p \in I$ is visible from r.

To visualise Polyanya, we employ a composite object comprising a circle primitive (representing r) and a polygon primitive formed by the points a, b and r. The polygon communicates the set of points visible from r through which an optimal path needs to pass on the way to the target. See Posthoc's website for an example.

Dual Euclidean Path Search on an Obstacle Map Dual Euclidean Path Search (DPS) (Hechenberger et al. 2022) is a novel method for constructing the shortest-Euclidean-distance path between two points on an obstacle-set map. In this data structure, obstacles are stored as polygons, not necessarily convex (Harabor, Hechenberger, and Jahn 2022). As such, new techniques like DPS are required to traverse it. For a problem navigating from point A to B, DPS initialises with a tentative solution of a straight path between A and B, ignoring all obstacles. Then, obstacles that intersect the current solution are iteratively introduced and the path is modified to tautly bend around them. When the path no longer intersects any obstacles, DPS has found an optimal path.

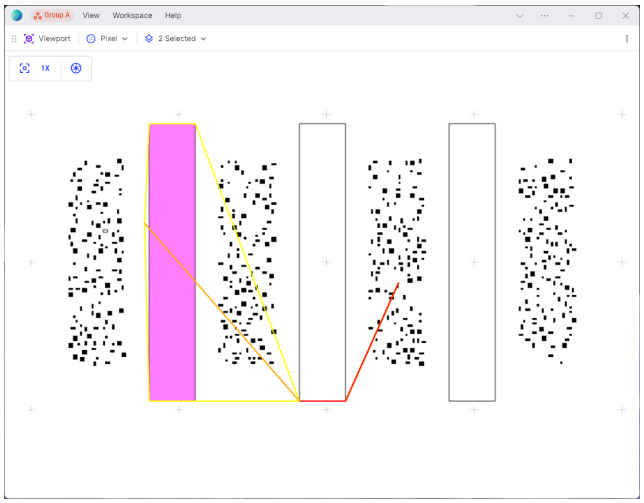

Figure 6: Dual Euclidean Path Search on an obstacle-set map.

A researcher used Posthoc to visually verify that an implementation of DPS is correctly bending paths around obstacles. They instrumented their C++ solver to output search traces. To visualise partial states, the researcher makes use of the `path` and `polygon` primitives as well as the `$for` and `clear` properties, highlighting valid and invalid segments of the path in green and red, and the currently processing obstacle as magenta (Figure 6).

Multi-agent Path Planning with CBS Multi-agent path planning concerns efficiently routing multiple agents under certain kinematic constraints. Conflict-based Search (CBS) (Sharon et al. 2021) is an optimal method for resolving agent collision, whereby a high-level search iteratively processes and resolves collisions between pairs of agents. In this synthetic example, we model and visualise the high-level search process in CBS as a *chronology* in the *graph* view. The viewport displays—for each node in the high-level search—the tentative paths for each agent and remaining unresolved collisions. See Posthoc's website for more multi-agent examples.

Debugging an Implementation of an Algorithm

Posthoc effectively helps researchers understand and debug problems. Here, a researcher was working to further improve runtime performance of weighted jump point search (JPSW) (Carlson et al. 2023) via pruning. In rare cases the implementation incorrectly pruned optimal solutions from the search space. Direct inspection of output logs did not produce any clues about the cause of the error. Using Posthoc, they created a visualisation of the search (Figure 7, left). Visual inspection of the search process allowed the researcher to pinpoint where and when the search began behaving unexpectedly and fix the logic error in minutes.

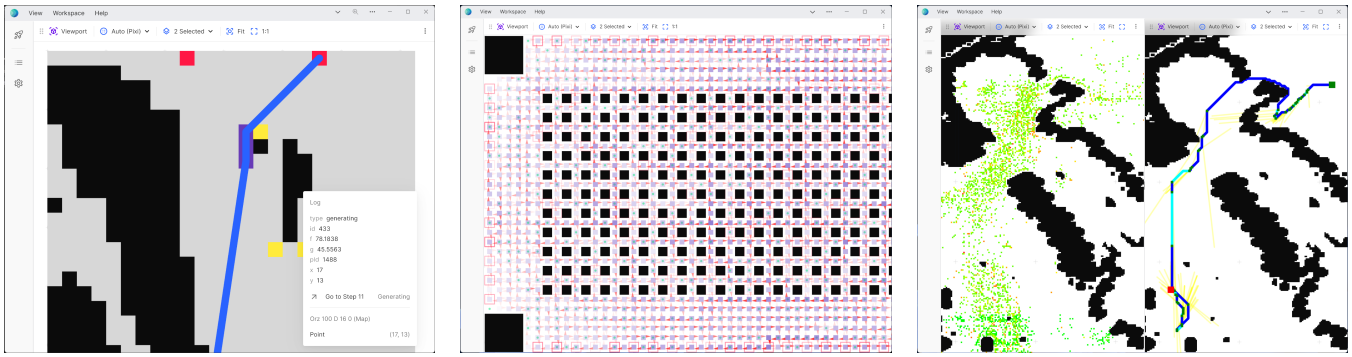

Figure 7: Various real-world usage examples. Left to right: JPSW on a grid map, traffic congestion heat map, StarCraft game analysis.

Optimising a Multi-agent Algorithm

Priority Inheritance with Backtracking (PIBT) (Okumura et al. 2019) is a recently proposed, performant, suboptimal technique for multi-agent path planning. Here a researcher used Posthoc to better understand the effectiveness of various PIBT variants on a medium-sized warehouse with 800 agents (Figure 7, middle). The visualisation, showing congestion, grid occupancy, and directional flows as a heat map, allowed them to identify opportunities for further improvement.

Game Analysis

Here a researcher wanted to analyse the behaviour of path-planning agents in a game of StarCraft. The game involves many agents and thousands of path planning episodes among numerous dynamic obstacles. The researcher produced several visualisations (Figure 7, right), including trajectories for each individual agent, heatmaps of all paths, and all locations appearing as a start or target. Posthoc remains resiliently performant in this complex example.

Education

Piglet (Chen et al. 2024) is a search solver written in Python with the primary purpose of teaching computer science students search algorithms. The Piglet team integrated *search trace* output into this solver as an example of how students could use visualisations in Posthoc to aid their understanding. In the simplest form, students were able to run a command to produce a search trace, which can then be imported into Posthoc to see a visualisation of the search process. This included depth-first search (DFS), Dijkstra's, breadth-first search (BFS), iterative deepening DFS (IDDFS), and A*. Since the visualisations were based on logging statements, it is trivial to add or adjust these statements to create visualisations that show different information. For example, the A* search in Piglet shows events like `dominated-by` and `relax`.

Conclusion

In this work, we contribute Posthoc, a framework that accelerates visualisation creation for the discipline of planning. We describe the *search trace format*, an easy-to-author format for recording program behaviour and defining relevant visualisations. We also demonstrate the Posthoc visualiser, a unified, comprehensive, and intuitive web environment for visualising and interacting with such search traces. We elucidate how such traces can be used to describe a variety of problems, processes, and domain-specific representations. Furthermore, we demonstrate how the Posthoc visualiser can be used in a range of contexts, in a variety of ways, to inspect, debug, and understand search algorithms. Finally, we validate its efficacy in a range of different real-world and synthetic case studies. Posthoc empowers researchers in planning and scheduling to incorporate visualisations into their workflow: to find and debug problems, come up with ideas and insights, collaborate with others, and share their work with the wider community. Posthoc transforms the planning discipline—by bridging the gap between complex data and intuitive understanding, Posthoc paves the way for groundbreaking innovations and collaborative breakthroughs.

Future Work

We identify two main areas of future work: (1) implementing renderers for specific domains, such as a 3D renderer, which may have a different set of primitives. A related piece of future work: a library of higher-order views, or the introduction of in-visualiser-editing, may further lower the barrier to creating visualisations. (2) enhancing support for solver-visualiser interactivity. Learners of search, planning, and scheduling would greatly benefit from visualisations of the latest ideas, techniques, and algorithms from this space. We invite researchers to integrate, via our API, their solver with our visualiser. That is, to be able to invoke the solver from the visualiser or vice versa—to explore what-if scenarios or quickly inspect solutions for related problem instances.

Acknowledgments

We wish to thank the students that have contributed to various prototypes that led to the design of Posthoc: Karan Batta, Jay Wingate, Surayez Rahman, Can Wang, Rory Tobin-Underwood, Francis Anthony, Bennett Madavana and Evelyn Huang.

References

Andreasen, M. Z.; Holler, P. I.; Jensen, M. K.; and Albano, M. 2022. MAES: A Realistic Simulator for Multi-Agent Exploration and Coverage. In *Sysem Demonstration, 32nd International Conference on Automated Planning and Scheduling (ICAPS)*.

Bast, H.; Delling, D.; Goldberg, A.; Müller-Hannemann, M.; Pajor, T.; Sanders, P.; Wagner, D.; and Werneck, R. F. 2016. Route planning in transportation networks. *Algorithm engineering: Selected results and surveys*, 19–80.

Bauer, A.; Botea, V.; Brown, M.; Gray, M.; Harabor, D.; and Slaney, J. K. 2010. An Integrated Modelling, Debugging, and Visualisation Environment for G12. In *Proceedings of the International Conference on the Principles and Practice of Constraint Programming (CP)*, 522–536.

Borissova, D.; and Mustakerov, I. 2015. E-learning tool for visualization of shortest paths algorithms. *Trends Journal of Sciences Research*, 2(3): 84–89.

Botea, A.; Bouzy, B.; Buro, M.; Bauckhage, C.; and Nau, D. 2013. Pathfinding in games. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Carlson, M.; Moghadam, S. K.; Harabor, D. D.; Stuckey, P. J.; and Ebrahimi, M. 2023. Optimal Pathfinding on Weighted Grid Maps. *Proceedings of the AAAI Conference on Artificial Intelligence*, 37(10): 12373–12380.

Chen, Z.; Harabor, D.; Nobes, T.; and Zheng, K. 2024. PIGLET: A Solver for Pathfinding Search. https://github.com/ShortestPathLab/piglet.

Cui, M. L.; Harabor, D.; and Grastien, A. 2017. Compromise-free Pathfinding on a Navigation Mesh. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*.

De Pellegrin, E.; and Petrick, R. P. A. 2024. Planning Domain Simulation: An Interactive System for Plan Visualisation. *Proceedings of the International Conference on Automated Planning and Scheduling*, 34(1): 133–141.

Guttman, A. 1984. R-trees: a dynamic index structure for spatial searching. *SIGMOD Rec.*, 14(2): 47–57.

Harabor, D. 2024. The Warthog Pathfinding Library. https://bitbucket.org/dharabor/pathfinding.

Harabor, D.; Hechenberger, R.; and Jahn, T. 2022. Benchmarks for pathfinding search: Iron harvest. In *Proceedings of the International Symposium on Combinatorial Search*, volume 15, 218–222.

Harabor, D. D.; and Grastien, A. 2014. Improving Jump Point Search. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, 128–135.

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

Hechenberger, R.; Stuckey, P. J.; Le Bodic, P.; and Harabor, D. D. 2022. Dual Euclidean Shortest Path Search (Extended Abstract). *Proceedings of the International Symposium on Combinatorial Search*, 15(1): e21787.

Ikeda, T.; Hsu, M.-Y.; Imai, H.; Nishimura, S.; Shimoura, H.; Hashimoto, T.; Tenmoku, K.; and Mitoh, K. 1994. A fast algorithm for finding better routes by AI search techniques. In *Proceedings of VNIS'94 - 1994 Vehicle Navigation and Information Systems Conference*, 291–296.

Kavraki, L. E.; and LaValle, S. M. 2016. Motion planning. In *Springer handbook of robotics*, 139–162. Springer.

Magnaguagno, M.; Pereira, R.; Móre, M.; and Meneguzzi, F. 2017. WEB PLANNER:: A Tool to Develop Classical Planning Domains and Visualize Heuristic State-Space Search. In *ICAPS Workshop on User Interfaces and Scheduling and Planning (UISP)*.

Okumura, K.; Machida, M.; Défago, X.; and Tamura, Y. 2019. Priority Inheritance with Backtracking for Iterative Multi-agent Path Finding. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, 535–542. International Joint Conferences on Artificial Intelligence Organization.

Park, J. J. 2020. Algorithm Visualizer. https://algorithm-visualizer.org.

Patel, A. 2020. Red Blob Games. https://www.redblobgames.com.

Rabin, S. 2019. *Game AI Pro 360: Guide to Movement and Pathfinding*. CRC Press.

Sánchez-Torrubia, M. G.; Torres-Blanc, C.; and Lopez-Martinez, M. A. 2009. PathFinder: A visualization eMath-Teacher for actively learning Dijkstra's algorithm. *Electronic Notes in Theoretical Computer Science*, 224: 151–158.

Sharon, G.; Stern, R.; Felner, A.; and Sturtevant, N. 2021. Conflict-Based Search For Optimal Multi-Agent Path Finding. *Proceedings of the AAAI Conference on Artificial Intelligence*, 26(1): 563–569.

Shishmarev, M.; Mears, C.; Tack, G.; and De La Banda, M. G. 2016. Visual search tree profiling. *Constraints*, 21(1): 77–94.

Simonis, H.; Davern, P.; Feldman, J.; Mehta, D.; Quesada, L.; and Carlsson, M. 2010. A generic visualization platform for CP. In *International conference on principles and practice of constraint programming*, 460–474. Springer.

Sturtevant, N. 2020. Moving AI: Single Agent Search Demo. https://www.movingai.com/SAS.

Vallati, M.; and Kitchin, D. 2020. *Knowledge Engineering Tools and Techniques for AI Planning*. ISBN 978-3-030-38560-6.

Wilkins, D. E. 2014. *Practical Planning: Extending The Classical AI Planning Paradigm*. Elsevier.

Xu, X. 2020. Pathfinding.js. https://qiao.github.io/PathFinding.js/visual.