Path Planning with Inventory-Driven Jump-Point-Search

Davide Aversa
Department of Computer, Control, and Management Engineering
Sapienza University of Rome
Rome, Italy
aversa@dis.uniroma1.it

Sebastian Sardina*
School of Computer Science and Information Technology
RMIT University
Melbourne, Australia
ssardina@cs.rmit.edu.au

Stavros Vassos
Department of Computer, Control, and Management Engineering
Sapienza University of Rome
Rome, Italy
stavros@dis.uniroma1.it

Abstract

In many navigational domains the traversability of cells is conditioned on the path taken. This is often the case in videogames, in which a character may need to acquire a certain object (i.e., a key or a flying suit) to be able to traverse specific locations (e.g., doors or high walls). In order for non-player characters to handle such scenarios we present InvJPS, an "inventory-driven" pathfinding approach based on the highly successful grid-based Jump-Point-Search (JPS) algorithm. We show, formally and experimentally, that the InvJPS preserves JPS's optimality guarantees and its symmetry breaking advantages in inventory-based variants of game maps.

Introduction

Pathfinding is a fundamental problem that arises in many diverse scenarios ranging from robots in the real world, e.g., (Bruce and Veloso 2002), to videogame characters in virtual worlds, e.g., (Botea, Mller, and Schaeffer 2004), (Björnsson and Halldórsson 2006). In all cases, an autonomous (physical or virtual) agent needs to reason over a map and find out how to reach a desired destination.

In videogames, in particular, it is often necessary to consider different capabilities for the characters that are performing pathfinding. For instance, some characters may be able to climb walls, fly, or swim, while others may not. Still, in those cases, the capabilities of the agents are *fixed*. Consider instead the scenario in which a non-player character (NPC) in a videogame is chasing the human player; the player goes through a corridor and locks the door behind him. With the corridor locked, the NPC has no other way to reach the player; nonetheless, there is a key, reachable by the NPC, that can unlock the door. Clearly, the NPC can reach the player if it first gets to the location of the key, picks it up, and then *goes back* to the corridor to open the door. Regular pathfinding approaches would fail to find a path (as the door is locked and there is no other way to reach the player). Common solutions in games involve hard-coding the behavior of the character to reach for a particular object (e.g., a key or suit), thus sacrificing flexibility at run-time as well as increasing the effort for developing the intended interaction.

*This work was conducted while a Visiting Professor at Sapienza Universita di Roma, Rome, Italy.

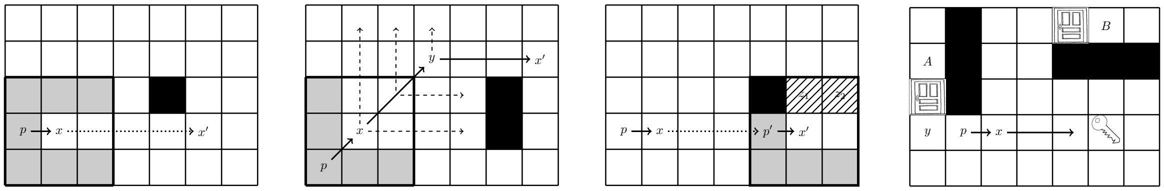
The issue with the above example is that the agent's capabilities (in the broad sense of allowing them to traverse locations) are *not* fixed at the outset, but *depend on the path that is followed by the agent*. Technically, whether a location is "blocked" or not, depends on whether the agent has visited a given location before and has *acquired then certain items or capabilities*. We shall refer to this variant of path planning as *inventory-driven pathfinding*.

Of course, one possibility for handling such settings is to employ general task-planning methods, such as classical STRIPS/PDDL planning (Ghallab, Nau, and Traverso 2004). By doing so, navigation, collecting objects, and using objects to open a blocked location, can be easily represented as actions with appropriate precondition and effects. However, as we will show, this turns out to be an impractical approach to the task and is not suitable for applications like videogames, for which pathfinding requests ought to be resolved almost instantly. We shall also point out that we do not aim at developing a complete new pathfinding approach for the inventory-driven task, but to rely on existing successful techniques as much as possible—*the simpler, the better*.

Our proposal is to build upon one of the most successful pathfinding algorithms, namely, Jump-Point-Search (JPS) (Harabor and Grastien 2011; 2012; 2014), and parsimoniously extend it to handle inventory-driven scenarios without losing its optimality properties nor, hopefully, its practical performance. More concretely, we present InvJPS, an "inventory-driven" variant of JPS that preserves the symmetry breaking advantages of JPS in the extended setting. We evaluate the approach over synthetic and real videogame maps that are augmented with doors and keys.

Planning via Jump Points

Jump-Point-Search (JPS) is a recent technique introduced by Harabor and Grastien in (2011) and further elaborated in (2012; 2014), that has proven extremely successful to navigate uniform cost grid-based maps. JPS is ultimately a speedup approach based on the well-known A* (P. E. Hart and Raphael 1968), a heuristic best-first search algorithm to find a minimum-cost path through a graph. Below, we shall assume that the reader is familiar with A* and describe the working of JPS to understand our contribution.



(a) Straight movement from x towards jump-point x' (see (c)).

(b) Diagonal movement from x. Node y is a jump-point.

(c) Two forced neighbours z_1 and z_2 of jump-point x'.

(d) A scenario where one key can open two doors.

Figure 1: Four scenarios describing JPS's straight and diagonal jumping mechanism and a key-door situation.

Overview JPS operates by identifying every node that might end up being on the optimal path, i.e., the so-called *jump-points*, and discards the rest. Thus, instead of expanding all reachable nodes, JPS jumps from a potential turning (jump) point to another turning (jump) point, expanding only those nodes that might require a change of direction.

Technically, JPS operates like A* by working through the grid systematically, maintaining an open list, and selecting and opening nodes from that list using the same best-first evaluation function as A*; but JPS is also a beam search in that it prunes as it goes. When a node x in the open list is expanded, A* retrieves *all* its unblocked adjacent nodes and adds them to the open list. Instead, JPS retrieves only some adjacent nodes of x, constructs a *vector of travel* from x through each of them, and identifies the first so-called "*jump point*" that occurs along that vector—a final successor of x—which is finally added to the open list. Informally, a jump point represents a location in which the traveling direction in the optimal path may change. Since intermediate nodes from node x to each jump point are not explicitly handled or stored (in the closed list), once the goal is found and the path needs to be assembled, JPS must "fill in the gaps" between jump-points to generate a straight or diagonal path.

Pruning Mechanism Given a node x reached via a parent node p, JPS prunes from any node $n \in$ *neighbours*(x) of x such that (assuming corner-cutting is not allowed) either:

1. there exists a path π' from p to n that does not go through x and is shorter than path $\pi = \langle p, x, n \rangle$; or
2. there exists a path π' from p to n that does not go through x, has the same length as $\pi = \langle p, x, n \rangle$, but π' has a diagonal move earlier than π.

For example, in Figure 1(a) and 1(b), pruned neighbours of x are marked in gray; the remaining neighbours of p are marked as white and are referred to as the *direct successors* of node x. The *natural successors* of x are the direct successors if one were to assume no obstacles whatsoever. In the absence of obstacles, non-natural successors are pruned. But the presence of obstacles may preclude the pruned rules to discard some non-natural successor, which therefore end up in the set of direct successors. Such non-natural successors that could not be pruned form the set of *forced neighbours* of x. In Figure 1(c), the patterned cells z_1 and z_2 are forced neighbours of x': they are not natural successors of x', but they could not be pruned because the move $\langle p', z_1 \rangle$ cuts corner and is hence not legal. Importantly, only straight movements may produce (up to 4) forced neighbours.

Jumping Procedure As explained by (Harabor and Grastien 2012), JPS applies to each actual successor–natural or forced neighbour—of the current node x a simple "jumping" procedure to replace each neighbour n with an alternative successor n' that is further away, the next "jump point." Technically, a *jump point* is a node that contains a forced neighbour. In Figure 1(c), node x' is a jump point. Intuitively, the fastest way to reach z_1 and z_2 from p' is via x', and hence node x' becomes a "turning point."

When moving in a straight manner, the only natural neighbour is explored recursively, in the corresponding straight direction, until either an obstacle, a jump point (i.e., a node having a forced neighbour; e.g., node x' above), or the goal is encountered. In the first case, the path is deemed failed, all nodes in it are ignored, and nothing is generated. If, however, a node n' having one or more forced neighbours—jump points—or being the actual goal is reached, then n' is generated as a next successor of x and is added to the open list; effectively "jumping" from x to n' without adding any of the intermediate nodes into the open list. So, in Figure 1(c), the final successor of node x is jump point node x', which has nodes z_1 and z_2 as a forced neighbours.

When moving diagonally, as in Figure 1(b), node x has three natural neighbours (white cells in x surroundings): two straight (north and east) and one diagonal (east north). JPS then recourses over the diagonal neighbour only if both straight neighbours produce failed paths. When there is a non-failed straight jump, then the node in the diagonal path is also considered an (indirect) jump point—a potential turning point—and is added to the open list (node y).

By jumping, JPS reduces memory consumption and the number of operations required. JPS was shown to improve on the fastest pathfinding approaches (including HPA*) by several factors in some cases (Harabor and Grastien 2011). JPS operations are all done online, with no pre-processing or memory overhead, and moreover, it is provably optimal.

Inventory-driven Path Planning

Here, we are interested in path planning *in the context of agents who carry objects*—an "inventory"—*that can influence the navigation process*. In such settings, reaching the destination may depend on whether the agent has acquired a certain object, such as a key to open a door, or a swimming suit or boat to pass across water, as is typical in many videogame worlds. We will use "items" to refer to objects or capabilities that can be acquired by the agent, and can be used to traverse or open some blocked nodes in the map.

We define *inventory-driven pathfinding* as follows. Given:

- a grid-based map M as a set of nodes $\{m_{11}, m_{12}, \ldots\}$;
- a set $O \subseteq M$ of nodes that are blocked (and cannot be traversed by the agent);
- a function *adj* : $M \mapsto 2^M$ denoting the adjacency relation among nodes (*adj*(x) denotes the set of nodes that adjacent to node x);
- a set of items $\mathcal{I}$ (e.g., objects or capabilities) that may be scattered in the map (and that the agent is able to acquire when co-located);
- a function *obj* : $M \mapsto 2^{\mathcal{I}}$ stating the items present in each location (*obj*$(x) = \emptyset$ denotes no items at node x);
- a function *req* : $M \mapsto 2^{\mathcal{I}}$ stating which items are required to traverse a node; and
- a start node S and destination node G,

find an *optimal (i.e., shortest) path* $\sigma = x_1, x_2, \ldots, n_n$, with $n \geq 1$, such that *(i)* $x_1 = S$; *(ii)* $x_n = G$; *(iii)* $x_{i+1} \in$ *adj*(x_i) and $x_i \notin O$, for every $i \in \{1, \ldots, n-1\}$; and *(iv)* $\bigcup_{j<i}$ *obj*$(x_j) \subseteq$ *req*(x_i), for every $i \in \{1, \ldots, n\}$ That is, we are interested in finding the shortest path from start to destination under the constraint that some locations along the path may require the agent to have previously visited other special nodes (fourth constraint above).

We note that this is a *one-shot complete-knowledge* task, in the sense that the agent has all the relevant information at the outset—the map nodes and their connectivity, the blocked nodes, the items at nodes, and the items that open blocked nodes—and will deliberate *offline* in order to find the shortest path that brings her to destination. Different strategies for re-planning may be employed to account for a dynamic map or other variants for incremental planning, e.g.,(Koenig and Likhachev 2005; Koenig, Likhachev, and Furcy 2004), but these are out of the scope of this work.

While there are various straightforward solutions for dealing with pathfinding in videogames in the context of characters with different capabilities, e.g., characters that can fly or swim, and characters with different sizes, there is no documented approach for dealing with this type of inventory-driven pathfinding where objects and capabilities can be acquired in the course of executing the path. Similarly, in the academic literature for pathfinding search, there is no approach, to the best of our knowledge, that handles this particular variant of pathfinding. Of course, inventory-driven pathfinding can be formalized as a planning problem, e.g., by appealing to classical STRIPS planning (Ghallab, Nau, and Traverso 2004), and using actions with appropriate precondition and effects in order to represent navigation, collecting objects, and using objects to open a blocked location. However, as we will show, being a domain-independent approach, this is an impractical approach for this task and is not suitable for applications like videogames, for which pathfinding requests ought to be resolved almost instantly.

Finally, we note that it is not possible, in general, to know in advance if an inventory item in the map will be needed or not in order to traverse the optimal path. This is indeed something to be discovered as part of the planning process and, as such, no item may in general be ignored at the outset.

Inventory-driven Jump-Point-Search

As we are motivated by videogame worlds, we looked into academic techniques that are most influential in the practical videogame setting. JPS is an award-winning algorithm and has attracted much attention within the game community, and as all other pathfinding approaches is not able to cope with such inventory-driven scenarios: it will either yield the best path to the goal that does not resort to acquiring extra capabilities or output no solution. In this work then we show that JPS can be further elaborated in a principled way to accommodate inventory-driven path planning. The new algorithm, which we call *inventory-driven JPS* (InvJPS) is obtained by modifying JPS in three simple ways.

The **first modification**, as one would do with any search approach, involves extending the state representation to account not just for the location of the agent, but also the current inventory. So, for a map M a state is a pair $\langle x, I \rangle$ where x is a node in M and I is a subset of elements from set $\mathcal{I}$ of all possible items. As in this work we do not consider the cost for obtaining or carrying an item, during search when an agent is at a location that has items, these are placed all in their inventory instantly. The items that an agent carries allow them to traverse also nodes that are marked as blocked but are labeled by *req* with a set of items such that the intersection with the inventory I is non-empty.

The **second modification** to JPS involves treating any node containing some capability or object as an "intermediate" goal. In other words, during the (recursive) jumping process, when an *intermediate* node x is generated such that *obj*$(x) \neq \emptyset$, then the process is deemed complete and x is considered a jump point (and thus added to the open list). We call such jump point nodes, *inventory jump points*. Note that when an item i is already contained in the state then acquiring another instance of the same item i (by visiting a node where the second instance lies) does not change the state representation in the search process and does not generate an inventory jump point. In the case though that two items i, i', such that $i \neq i$ happen to unlock the same nodes in the map, acquiring the essentially "duplicate" item i' is handled in the generic way by the algorithm. Finally, observe that, contrary to what one would expect, we do not require any change to the pruning mechanism and hence we took a "lazy" approach to inventory "finding."

In classical JPS, when a new (jump point) node is retrieved from the open list for expansion, *all* directions towards natural and forced neighbours of the node in question are considered for further "jumps." The natural successors represent the same direction the agent was traversing when a jump point was found, whereas the forced neighbours represent the turning directions that the agent may need to consider (see (Harabor and Grastien 2011, Lemma 1)). This implies, for example, that it is not necessary to consider the parent of the jump node, as this would involve undoing the path traveled so far. However, in the context of inventory-based path planning, if the node being expanded is an inventory jump point, then the agent ought to consider *all possible* *directions*, including that undoing the path traversed so far! The fact is that, with the new inventory acquired, nodes that looked "blocked" before may have now become traversable.

So, the **third modification** included in InvJPS is to treat inventory jump point nodes as the starting node, thus applying the jumping process towards all possible directions. This is depicted in Figure 1(d) (assume the key opens all doors). When jumping east from x, the node with the key becomes a jump point. From there, the agent must consider jumping towards *all* directions, not just east. In fact, the agent should consider returning back to node x as a new jump point, now though holding the key. Later, the agent will possibly generate node y as a jump point too, because the node north to it is now open to the agent and will allow her to visit area A. A similar argument can be given for the north door and area B (using the north-west direction).

Let us refer with InvJPS to the algorithm obtained from implementing the above three modifications to JPS. The following result states that the optimality of JPS is preserved.

Theorem 1. *InvJPS always returns an optimal solution for any inventory-driven path planning problem. If there is a path from start to destination, InvJPS returns a solution.*

Proof (Sketch). This can be proved by following the same reasoning as the proof of (Harabor and Grastien 2011, Theorem 1), but inductively accounting for each key required along the arbitrary optimal path selected, and treating each inventory node in that path as an "intermediate" goal. We rely then on JPS optimality, to claim that we get to all intermediate goals optimally (and eventually to the goal). The base case for the induction is when the optimal path to goal contains no inventory nodes, and hence, the standard reasoning for JPS is valid (see Theorem 2 below). □

In fact, it is easy to see that when applied to non-inventory path planning problems, InvJPS reduces to regular JPS.

Theorem 2. *If InvJPS is applied to a path planning problem where $obj(x) = \emptyset$, for all $x \in M$, then InvJPS generates the same search space than JPS, including the same list and order of nodes in the open list and intermediate expansions.*

Proof. As there are no nodes with objects in the map M then: *(i)* all search states are of the form $\langle x, \emptyset \rangle$, hence the cost of navigating through a cell remains unchanged; and *(ii)* no inventory jump point nodes are ever generated. □

Analysis of performance challenges for InvJPS

For simplicity, we will refer to inventory nodes as nodes with *keys* that open special (locked) *door* nodes. A central observation is that, when InvJPS reaches a key, it considers going "backwards," thus re-exploring previously explored areas in the hope of finding a now open door. Informally, the worst case scenario for InvJPS arises when the whole map is covered with keys such that each node contains a different key and the goal is unreachable, in which case the algorithm has to search the whole map multiple times, once for each possible combination of keys. However, this kind of scenario is unikely to be seen in real games. Instead, we investigate how the number of keys in the map, their placement, and whether those keys are necessary for the optimal path, play an important role in how much "re-exploration" is performed.

Consider a map with only one key k that is located at node x_k, and start and destination nodes x_s, x_d. Suppose first that there are no doors in the map, and so the key is not necessary for finding a path that reaches the destination. In such case, InvJPS faces a performance drawback scenario in the sense that one could simply apply standard JPS. Motivated by this, we report below on experiments with redundant keys to evaluate the (sometimes) unnecessary overhead of InvJPS over JPS. Note however that this is somehow an unfair comparison, because the whole point is that one does not know at the outset whether the keys are necessary or not.

Now, suppose further that the search reaches node x_k (where the key is) and so generates the search state $\langle x_k, \{k\} \rangle$. As all directions from there will be explored, the search could eventually be exploring concurrently search states of the form $\langle y, \{\} \rangle$ and $\langle y, \{k\} \rangle$, that is, the same location node y but with different inventories (and different g cost). However, the key is actually not needed. In this case, the *earliest* that x_k is encountered (e.g., if it is closest to the starting location), the *more overhead* it will impose.

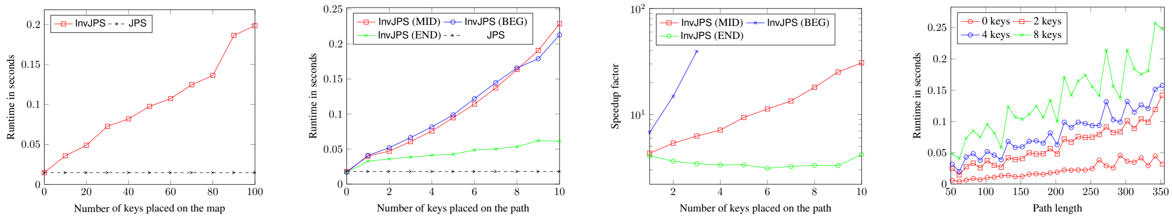
Now consider a map with only one key k as before, but in this case there is also a locked door node y such that all paths to the destination *necessarily* pass through y. In this case the fact that $\langle x_k, \{k\} \rangle$ is added to the open list is actually helpful, and it is more beneficial to happen early in the search process. As before, there will be nodes that are checked twice in different states of the form $\langle y, \{\} \rangle$ and $\langle y, \{k\} \rangle$, even though the first is not necessary as there is no way to reach the destination without the key. Observe though that, unlike the cases where the key was *not* needed, here the *latest* that x_k is encountered (e.g., if it is farthest from from the starting location), the *more overhead* it will impose. For instance, consider the case in which the search needs to explore *all the search space* before hitting the node with the key; then the search will essentially start over from $\langle x_k, \{k\} \rangle$ to reach the destination going over nodes that have been explored in states with an empty inventory.

Finally, note that this scenario we described for the case that key k is needed, is in fact very similar to the case in which a regular pathfinding search fails to find a path after exploring the whole search space. This suggests that we can test for "near-worst case scenarios" when keys are needed, by means of checking InvJPS on *unreachable* paths over maps with keys. This is clearly an upperbound for the expected overhead but also gives some qualitative insight.

Experimental Analysis

We report here on a series of experiments that demonstrate the performance of InvJPS applied to videogame maps from the Moving AI benchmark[1] (Sturtevant 2012) as well as synthetic maps we built. We use a Python implementation of the algorithms and run the experiments on an Intel i7 3.2GHz machine (on a single core) with 8GB of RAM. The runtime of the algorithms reported are often higher than the one that

[1] http://movingai.com/benchmarks/



(a) Runtime of algorithms with keys randomly placed on the map.

(b) Runtime of algorithms with keys placed on the path.

(c) Speedup for the cases of (b) compared to InvA* (log scale).

(d) Runtime of InvJPS with keys at the beginning of the path.

Figure 2: Experimental results for the case that keys are not necessary over benchmark game maps.

can be achieved by C++ implementations that would typically be employed in videogames. Nonetheless, the particular implementation platform does not alter the core findings and conclusions and, of course, one would always resort to the most efficient platform at deployment time.

In the first three experiments, we used the benchmark consisting of maps from "Baldurs Gate II" that are scaled to 512×512 nodes. From each of the 75 maps we took 50 pathfinding instances that are known to be realizable and have a length between 150 and 250 steps.

Experiment 1: Random placement of *unnecessary* keys over *real* game maps; analysis per number of keys. First, we look into the case where regular maps are extended to include keys but not doors, so that the keys are *not necessary* (and every key is different one from the another), and we compare to running regular JPS. This experiment aims to evaluate the "theoretical" overhead of InvJPS over JPS. Effectively, the keys become "noise" for the planner, as discussed in the previous section.

To extend the standard key-free maps, we added $0 - 100$ distinct keys in a random way in each map and run InvJPS and JPS on those instances. The runtime of the algorithms is reported in Figure 2(a). As expected, there is an overhead for considering keys, even when not needed, which scales well (almost linearly) over large numbers of keys. In particular, the overhead of running InvJPS over for the case of having 100 keys on the map is equivalent to running approximately 10 regular pathfinding queries with JPS. This is because the runtime is affected only by the keys which are actually taken into account during search. As we spread the keys uniformly on the map, not every node containing a key will be explored by the algorithm. 100 keys is probably a large number for such maps, but it ensures that at least a few of those keys will appear in the random paths we try in this experiment.

Another way to interpret the results is that what we see is in fact the start of an exponential overhead with respect to the small number of keys that are actually encountered during search in these random scenarios. The next experiment intends to explore this further by placing the keys in a positions that are relative to the shortest path solution.

Experiment 2: Placement of *unnecessary* keys on the path over *real* game maps; analysis per number of keys. Here, we consider again unnecessary keys but, following the discussion in the previous section, we look into the case that they are selectively inserted in three areas: *(i)* at the beginning of the path (BEG); *(ii)* at the end of the path (END); and *(iii)* distributed evenly on the shortest path (MID). We assume that one tile can be occupied by at most one key, therefore, keys are scattered appropriately in a small area around the starting tile (BEG), the destination (END) or around the shortest path. We tested InvJPS, plain JPS, as well as InvA*, a straightforward inventory-driven variant of standard A*.

As argued, the overhead is expected to increase when more keys are reached during search. The runtime of InvJPS compared with that of JPS (for reference) is reported on Figure 2(b), while the speedup of InvJPS compared to InvA*, in terms of how much faster is the runtime of InvJPS, is reported on Figure 2(c). Note that every configuration (BEG, END and MID) is compared with InvA* in the same configuration, e.g., InvJPS (BEG) is compared to InvA* (BEG).

The results validate the expected effect of the placement of unnecessary keys on the runtime performance, with the (BEG) distribution being the worst configuration and the (END) distribution the best. The exponential speedup we observe in the BEG scenario is due to the known speed-up of JPS over A*, but applied over all those conditioned cases of the search space for every combination of keys that is encountered. As the number of these combinations is exponential to the number of keys, the speedup of InvJPS then is also exponential. In fact, when we explore a smaller number of conditioned cases such as in the END scenario the exponential speedup is less evident (note that Figure 2(c) is in logarithmic scale). These results validate that the benefits of JPS are carried over in the inventory-driven setting. In particular, in the case of just 4 keys close to the start location, InvJPS is more than 300 times faster than InvA* (InvA* is not able to cope with more than 4 keys here).

Experiment 3: Placement of *unnecessary* keys on the path over *real* game maps; analysis per path length. In this experiment we test InvJPS over the most challenging scenario from the previous analysis, i.e., the BEG scenario, with respect to the path length. We consider 100 realizable paths with no length restriction per map. We then compute the optimal path and construct inventory-driven instances where keys are placed in the beginning of the path. Figure 2(d) shows the runtime of InvJPS wrt plan length for the cases in which 0, 2, 4, and 8 keys are added in the beginning of the path. When no keys are added, InvJPS reduces to regular JPS (Theorem 2). This can be used as reference to see the impact of the number of keys for different path lengths, compared to non-inventory paths of the same length that are solved with JPS. Observe that there is an approximately linear increase of the runtime of InvJPS as the path length increases. While, as expected, more keys induce more effort to InvJPS, it is still able to solve problems with long paths, which would be impractical due to memory and time constraint for any A* version doing complete node expansions.

Experiment 4: Incremental scenario of *necessary* keys over *synthetic* maps; analysis per number of keys. Here, we focus on the case where keys are actually needed in order to reach destination. As there is no benchmark maps for this setting, we built synthetic 512×512 maps of two types starting from completely empty maps. Then, for each number of necessary keys ranging from 0 to 10, we report the average runtime of InvJPS over 200 pathfinding instances.

In the first type of maps, we build artificial rooms that separate the empty map from left to right by putting vertical walls and a door between adjacent rooms. We impose a "sequential" traversal of the rooms in the sense that door d_i cannot be reached without crossing door d_{i-1}, and key for door d_i can always be found in the map region before d_i. The destination is behind the last door. InvJPS's performance is shown in Figure 3(a). For the second type, we impose a "detour" behavior. For n keys, we start from the destination, build a "room" around it, and then create a random entry point with a door d_n. We then chose a random point in the map, place key k_n for door d_n, and repeat the process now with key k_n (until all keys and doors are configured). Unlike the first case, here doors are all reachable by the agent from the starting state, but solutions require the agent to go back and forth along all the map as the key for door d_i is closed in the room blocked by door d_{i-1} and so on. InvJPS's performance is shown in Figure 3(a).

As expected, in the "detour" scenario the average path length (450-500 steps) is larger than in the "sequential" scenario as well as the other experiments, due to the multiple detours the agent has to perform. As the placement of rooms is done randomly and we only place a small number of rooms, the whole area of the map is not necessarily covered though, and this is why the average path length is not larger than the map dimensions. Observe also that the runtime of InvJPS for 10 keys in the "sequential" scenario is comparable to the runtime of InvJPS in Experiment 2 with 10 unnecessary keys in the challenging "BEG" scenario. Finally, note that other approaches, including JPS, cannot be applied to find a path here as they will return no solution.

Experiment 5: Key performance overhead for unreachable destinations. Here, we test InvJPS over instances that are not solvable, which in fact identify the overall worst case scenario (as all the search space is explored), but also provide an upper-bound for the worst case when keys are needed, as discussed in the previous section. We produced 200 paths for keys from 1 to 10 over the synthetic maps of Exp. 4 but we surrounded the destination by walls. The results, reported in Figure 3, confirm a higher runtime than all the other experiments. For 10 keys, the runtime is approxi-

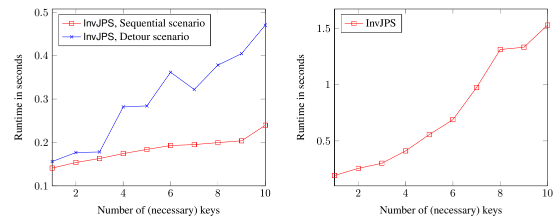


Figure 3: Performance of InvJPS when keys are necessary (left) and when no path exists (right).

mately 7 times more than the "BEG" scenario, while it starts to show the inherent exponential nature of inventory-driven pathfinding. Consider though that due to the symmetry-breaking benefits of InvJPS, reasoning about this scenario is nonetheless possible, while it is not feasible for InvA* to even handle beyond 4 keys. Also, as happens in practice in videogames, a high runtime for unreachable paths when more keys are present can be avoided by imposing practical upper-bounds on the resources, and sacrifice completeness.

General-purpose planning Finally, it is possible to solve inventory-driven pathfinding by using general action-based planning (Ghallab, Nau, and Traverso 2004). Such techniques in fact go beyond what InvJPS can handle, as they can mix path planning with more general dynamic reasoning, such as delivering packages or solving puzzles. However, the performance of such general approaches is *not competitive* for the particular case of inventory-driven pathfinding. We performed experiments with state-of-the-art planners FF (Hoffmann 2001) and LAMA (Richter and Westphal 2010) on inventory-free maps that confirm this. FF could only parse maps of size up to 120×120, and took a couple of seconds to synthesise paths of length below 100. LAMA on the other hand was able to parse maps of size 512×512 (as those used in our experiments with InvJPS), but could not solve many of those maps in several minutes. It is important to remark that this is hardly surprising, and not even a fair comparison, since such planners rely on domain-independent heuristics and are meant to be used when known domain heuristics are not available.

Related Work

There are several works for improving A* in path planning, e.g., RWA* (Richter, Thayer, and Ruml 2010), RTA* (Korf 1990), and DAS (Dionne, Thayer, and Ruml 2011). While they can all easily be extended to deal with inventory-driven path planning (by just including the inventory in the state), we expect they will all suffer the same degradation as A*, as too many ("richer" in form) nodes make it to the open list, in fact exponential to the number of keys in the worst case.

JPS, on the other hand, comes from a different tradition that is geared towards game programming and grid-based search with a focus on symmetry, state-space pruning, and pre-processing. Other works in this tradition are Near-Optimal Hierarchical Pathfinding (HPA*) (Botea, Mller, and Schaeffer 2004), Swamps (Pochter, Zohar, and Rosenschein 2009), and Rectangular Symmetry Reduction (RSR) (Harabor, Botea, and Kilby 2011). In their current form, none of these existing approaches address inventory-driven path planning, but can also be extended in a similar way. For Swamps it probably requires to treat inventory nodes as (intermediate) goals. For HPA*, one could consider handling the inventory only at some levels of abstraction. However, if the reasoning is done too low, completeness may be lost. We believe that their performance will not degrade in a dramatic manner when generalizing to inventory-driven domains, however, we expect the benefits of JPS over them—as reported in (Harabor and Grastien 2011)—to remain unchanged against the inventory-driven versions.

As far as videogames are concerned, even though they often feature rich dynamic worlds where the inventory-driven pathfinding scenario can easily arise, non-player characters (NPCs) are typically unable to use their surroundings in this way. In particular about doors and passages, what happens in practice is that either the game levels are designed in such way that this type of interaction is irrelevant, or the NPCs have some hard-coded rules for using special objects as dictated by game level designers (e.g., break into doors).

Conclusion and Future Work

In this paper, we introduced the problem of *inventory-driven pathfinding* where the traversability of nodes is conditioned on the path taken, and generalized Jump-Point-Search (JPS), one of the most competitive path-planning algorithms, to handle such scenarios. The proposed algorithm, InvJPS, motivates a middle ground approach for practical deliberation mechanisms that are similar to general action-driven planning techniques, but are also *grounded* to the performance achievements of sophisticated pathfinding approaches. To the best of our knowledge there is no prior work that addresses the problem of inventory-driven pathfinding in the literature. Here we showed that the proposed algorithm InvJPS is not only a parsimonious extension of JPS that can handle inventories, but it also preserves the performance and symmetry-breaking advantages of JPS.

There are several extensions and optimizations we intend to explore. One of the most immediate extensions is to lift the assumption that there is no cost for picking up an item and no limit on the number of items that the agent can carry. As far optimizations are concerned, there are two important categories. First, optimizations that preserve optimality by re-using information from previous search that has been performed with a smaller inventory, in order to avoid some double effort. This can be done in a similar way that incremental search re-uses part of the explored search space, as e.g. in "life-long planning A*" (Koenig, Likhachev, and Furcy 2004), but in a way such that only the information that is not affected by the new item can be reused. Second, optimizations that allow computational speedups in exchange for finding possibly sub-optimal paths. For instance a variant of InvJPS may deviate from the most promising path in order to pick-up keys that are "near" the current location. In this way, as taking a key can only increase the reachable areas of the map, by sacrificing some optimality for picking up also keys that may not be useful at the end, we simplify the combinatorics involved and we allow for faster solutions.

References

Björnsson, Y., and Halldórsson, K. 2006. Improved heuristics for optimal path-finding on game maps. In *Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*.

Botea, A.; Mller, M.; and Schaeffer, J. 2004. Near optimal hierarchical path-finding. *Journal of Game Development* 1:7–28.

Bruce, J., and Veloso, M. M. 2002. Real-time randomized path planning for robot navigation. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2383–2388. IEEE.

Dionne, A. J.; Thayer, J. T.; and Ruml, W. 2011. Deadline-aware search using on-line measures of behavior. In *Proceedings of the Annual Symposium on Combinatorial Search (SoCS)*, 39–46.

Ghallab, M.; Nau, D. S.; and Traverso, P. 2004. *Automated Planning: Theory and Practice*. Morgan Kaufmann Publishers Inc.

Harabor, D. D., and Grastien, A. 2011. Online graph pruning for pathfinding on grid maps. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*.

Harabor, D. D., and Grastien, A. 2012. The JPS pathfinding system. In *Proceedings of the Annual Symposium on Combinatorial Search (SoCS)*.

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.

Harabor, D. D.; Botea, A.; and Kilby, P. 2011. Path symmetries in undirected uniform-cost grids. In *Proceedings of the Ninth Symposium on Abstraction, Reformulation, and Approximation, SARA 2011*.

Hoffmann, J. 2001. FF: The fast-forward planning system. *AI Magazine* 2(3):57–62.

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

Koenig, S.; Likhachev, M.; and Furcy, D. 2004. Lifelong planning a. *Artificial Intelligence* 155(1-2):93–146.

Korf, R. E. 1990. Real-time heuristic search. *Artificial Intelligence* 42(2):189–211.

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

Pochter, N.; Zohar, A.; and Rosenschein, J. S. 2009. Using swamps to improve optimal pathfinding. In *Proceedings of the International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, 1163–1164.

Richter, S., and Westphal, M. 2010. The LAMA planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research (JAIR)* 39:127–177.

Richter, S.; Thayer, J. T.; and Ruml, W. 2010. The joy of forgetting: Faster anytime search via restarting. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, 137–144.

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