Hipster: An Open Source Java Library for Heuristic Search

P. Rodríguez-Mier, A. González-Sieira, M. Mucientes, M. Lama and A. Bugarín
Centro de Investigación en Tecnoloxías da Información (CITIUS)
University of Santiago de Compostela, Spain
{pablo.rodriguez.mier, adrian.gonzalez, manuel.mucientes, manuel.lama, alberto.bugarin.diz}@usc.es

Abstract— **In this paper we present Hipster: a free, open source Java library for heuristic search algorithms. The motivation of developing Hipster is the lack of standard Java search libraries with an extensible, flexible, simple to use model. Moreover, most of the libraries for search algorithms rely on recursive implementations which do not offer fine-grained control over the algorithm. Hipster provides a wide variety of classical search algorithms implemented in an iterative way like Dijkstra, A*, IDA*, AD* and more. In order to facilitate the use and integration with most research, commercial and non-commercial projects, the software is developed under the open source Apache 2.0 License. Hipster was successfully applied in two different research projects in the areas of Web service composition and motion planning. Source code, documentation, binaries and examples can be found at *https://github.com/citiususc/hipster*.**

Keywords: heuristic search; uninformed search; informed search; local search; Java library.

I. INTRODUCTION

Many areas in Computer Science, such as robotics, planning, bioinformatics or web intelligence use heuristic search techniques to provide efficient solutions to common problems. State space search is widely used in Artificial Intelligence to model and solve general and specific problems in which the search space is divided into states that represent a particular configuration of the problem [1][2]. In this model, the search is performed from an initial state —the initial configuration to the problem— to a goal state, applying different actions in order to find a solution to the problem. Algorithms are often classified into two categories, depending on the information they use. These categories are uninformed and informed search. Uninformed search refers to those algorithms that do not have information about what state to expand next. Examples in this category are Depth First Search, Breadth First Search, Dijkstra or Bellman-Ford among others. On the other hand, informed algorithms are those techniques that use problem-specific knowledge —usually heuristics— that are used to estimate the distance to the goal in order to improve the performance by reducing the number of explored states. Informed search can in turn be divided depending on whether they exploration is over the whole state space —global search— or only over a part of the search space —local search. Examples of the first category are: Best First Search (BFS) [2], A* [3], IDA* [4] or D* [5], including the specific path search algorithms with re-planning capabilities such as ARA* [6] or AD* [7], whereas most common local search strategies are Beam Search [2], Hill Climbing [2], Enforced Hill Climbing [8], Tabu Search [9] or Simulated Annealing [10].

One of the common problems is the lack of generic search libraries with a flexible model that can be directly applicable to any problem. This forces developers to program domain-specific solutions based on generic algorithms for each new problem. Furthermore the use of restrictive or viral licenses in the current libraries makes even harder to reuse generic algorithms.

As a response to these problems, in this paper we present Hipster, a generic Heuristic Search library for Java. Hipster relies on a flexible model with generic operators to change the behavior without modifying the internals. All algorithms are also implemented in an iterative way, avoiding recursion. This has many benefits: full control over the search, access to the internals at runtime or a better and clear scale-out for large search spaces using the heap memory. Hipster also comes with a permissive Apache 2.0 license that allows the library user the freedom to use and modify it for any purpose.

At its current state, Hipster implements the following algorithms of these families: uninformed search —DBS, BFS, Dijkstra and Bellman-Ford—, informed search —A*, IDA* and AD*— and local search —Hill Climbing (HC) and Enforced Hill Climbing (EHC). More implementations will be added in the near future, but it is easy for any user implement any other algorithm using the Hipster model.

II. PROJECT GOALS

The library was implemented following some guidelines to achieve the following goals:

Iterative algorithms. We implemented the algorithms in Hipster as iterative processes. The reason behind this is that most search libraries provide very simple interfaces which only require the initial and goal state, and once the search is executed the control is not recovered until the processing finishes. This makes unfeasible to modify the behavior of the algorithm —add new goals, extend the search process after finding the goal, etc.— and to monitor the search.

P. Rodríguez-Mier and A. González-Sieira are supported by FPU grants (refs. AP2010-1078 and AP2012-5712) from the Spanish Ministry of Education, Culture and Sports. This work was supported in part by the European Regional Development Fund (ERDF/FEDER) under the project CN2012/151 of the Galician Ministry of Culture, Education and Universities. Support from Spanish Ministry of Economy and Competitiveness under grants TIN2011-22935 and TIN2011-29827-C02-02 is also acknowledged.

Flexible, extensible, reusable. We are aware that solving a search problem requires to combine several pieces —state definition, transitions, the algorithm, a cost function, heuristics, etc.—, and in many occasions some of these pieces may change. The development of Hipster is based on the encapsulation of these pieces in separate components so each one can be changed without affecting the others, maximizing the flexibility of the model. It also facilitates to reuse and extend previously implemented components.

Powerful but simple API. The key insight the API of Hipster is to provide an easy way to solve simple search problems using the default components —which also reduces the barrier of entry for new users— and separate the most advanced options that allow the complete customization of the algorithms.

Highly-tested code. One of the project goals of Hipster is to provide well tested algorithms and hence a robust implementation. The quality of the code is guaranteed by automatic unit testing and continuous integration tools.

Permissive Apache 2.0 license. Most of the available Java libraries are released under restrictive licenses that are not compatible with many research and commercial projects. To minimize these conflicts and to maximize the adoption of the library, we chose a very permissive Apache 2.0 free software license. This license grants the freedom to use the library for any purpose, to modify it, and to distribute modified versions under the terms of the license, without concern for royalties.

III. Implementation Details

A structure of a search problem is formally defined by several components: the state space of the problem —all states reachable by applying any sequence of actions—, the transition model used to navigate between states, the initial state and, optionally, one or more goal states. Executing a search algorithm to solve the problem and find the optimal solution implies to build a search graph, where the nodes correspond to the states and the arcs are the transitions between them. The information generated during the search is stored in these nodes. Although different algorithms store different information in the nodes, these elements are always present:

- **State**: State in the state space that corresponds to the current node.
- **Parent**: Node in the graph that generated the current one.
- **Transition**: Action applied to move from the parent node to the current one.

This information is enough to execute simple algorithms like BFS or DFS. Nevertheless, most search algorithms evaluate the transitions to obtain the path with the minimum cost. This requires to store in each node the information about the **cost**, *g(n)*, of the path between the initial state and the current one. Moreover, algorithms that execute an informed search store an additional element, the **score**, which accumulates the cost from the beginning and the estimated cost to the goal according to a heuristic function, *h(n)*. Figure 1 shows how Hipster captures all these elements. *Node* is a generic interface that defines the common operations for all node types described above. An abstract implementation of *Node*, *AbstractNode*, is provided and can be extended to obtain custom defined node types to use with the search algorithms. The interface *Node* is extended by *CostNode* and *HeuristicNode*, which respectively manage the cost and score operations required by some algorithms, as described before. By default, Hipster uses two different implementations for the nodes, namely *SimpleNode* and *HeuristicNodeImpl*. These two implementations are enough for the included algorithms, but more sophisticated nodes can be created by extending *AbstractNode* without affecting the implementation of the algorithms. As all implementations extend the *Node* interface, the internal data type used in the algorithm is not required to be known, and the compatibility between them is guaranteed.

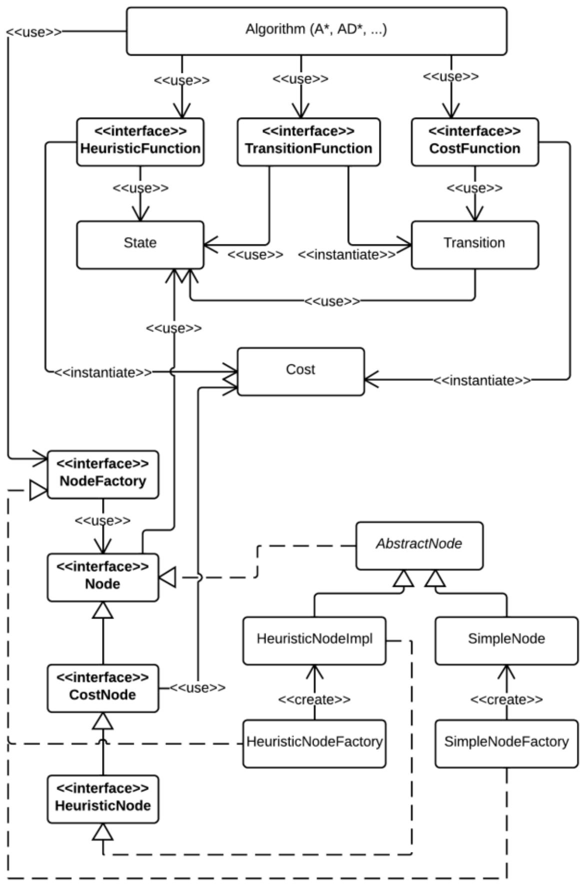

Figure 1. Data model of Hipster. The graph search is built using *States*, *Transitions* and *Nodes*, and the search algorithms may use heuristic and cost functions to evaluate the cost of the paths. The information generated during the search is stored in the *Node* elements, which are instantiated by a *NodeFactory* component.

Depending on the type of nodes, the algorithm uses an implementation of *NodeFactory* that instantiates the required node type. If the algorithm requires using *CostNode* elements, this component is also responsible of aggregating the cost from the beginning state associated to each node. Moreover, if *HeuristicNode* is used, it also computes the score by aggregating the cost of the node and the heuristic provided by the *HeuristicFunction* component. Hipster provides two different factories used for informed and uninformed search algorithms: *HeuristicNodeFactory* and *SimpleNodeFactory* respectively. These implementations are enough to work with most search strategies and it is possible to extend these components although it is not the most common scenario.

The algorithms implemented in Hipster rely on different components that encapsulate independent operations in the search process, such as generating the outgoing transitions of a state (*TransitionFunction*), evaluating the cost of the generated transitions (*CostFunction*) and estimating the cost between a state and the goal (*HeuristicFunction*):

- *TransitionFunction*: This interface provides a function that takes a state as an argument and returns its outgoing transitions, according to a transition model that depends on the problem.
- *CostFunction*: Evaluates a *Transition* object and returns its corresponding cost. The cost is a generic type which may be a numeric value or have a more complex definition.
- *HeuristicFunction*: Estimates the cost of the path between the input *State* object and the goal *State*. The type returned by this component must be consistent with the *CostFunction* implementation used so they can be aggregated.

As follows from Figure 2 Hipster uses a generic definition for the cost. This is motivated by the need of using complex definitions for the cost that can be a composition of several attributes that cannot be easily summarized in a single value. This is something that other search libraries do not contemplate and it is an important limitation when dealing with complex real problems. To manage an abstract definition for the cost we need to define the components to operate with the custom cost objects. In order to operate with these costs we define a binary function over the domain of the costs:

$$f : C \times C \to C \tag{1}$$

This corresponds with the interface *BinaryFunction* (Figure 2) that accepts two generic costs as input and returns the resultant transformation in the same domain, which allows to define generic operators as addition, scaling, etc. These are used in the *NodeFactory* to operate with the costs as detailed above. Hipster implements some common operators to work with *Double* cost types that are used, for example, to accumulate the costs when performing a Dijkstra search or to compute the score of a node as *g(n)* + *h(n)* in the A* algorithm.

In addition to the *BinaryFunction* interface we need to fulfill some properties to define a valid cost algebra [1] to work

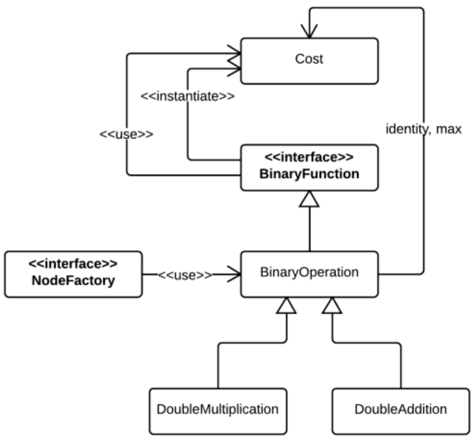

Figure 2. Class diagram showing the cost algebra used to manage custom type cost elements.

with generic costs. A cost algebra is defined as a 5-tuple $\{A, x, \preccurlyeq, C_{id}, C_{max}\}$ such that $\{A, x, C_{id}\}$ is a monoid, $\preccurlyeq$ is a total order between the elements of the domain and C_{id} and C_{max} are the identity and greatest elements of our cost. In order to define a monoid $\{A, x, C_{id}\}$, the binary function (x) over the domain (A) must be associative and have an identity element (C_{id}). Hipster provides implementations for the addition and multiplication of Doubles using this strategy, which are the most common scenarios in a great variety of search problems. It is worth to note that all these details are hidden to the user and it is not necessary to work at that level of abstraction unless a custom definition of the cost is used.

Following this design each component has a separate function. This encapsulation allows changing the implementation of each component without affecting the others. This facilitates reusing and extending them, and also makes the basic structure of the algorithms fully reusable.

The algorithms currently implemented in the library are divided in three different categories according to their features.

Uninformed algorithms:

- **Depth First Search (DFS)** [2]: It is a blind algorithm that performs an exploration of the graph in a way that always reaches the deepest node before backtracking. The Hipster implementation is a graph-based search that can handle cycles. It uses *SimpleNode* instances because it does not compute costs. This algorithm is complete (it always finds a solution if it exists) but not optimal (with the minimum cost).
- **Breadth First Search (BFS)** [2]: It is similar to DFS but in this case the exploration is done visiting all the successors of a certain level from the beginning state before going deeper. As DFS, it also uses *SimpleNode* instances and it is complete but not optimal.
- **Dijkstra** [1]: It is an optimal and complete graph search algorithm for non negative costs that visits the nodes in the order given by the minimum cost of the path from the beginning state. As it involves evaluating

the transitions between nodes, and tracks the cost from the start, it requires the usage of *CostNode* instances.

- **Bellman-Ford** [1]: It is an algorithm very similar to Dijkstra that can handle negative costs but at the price of a worse computational complexity.

Informed algorithms:

- **A*** [3]: It is a heuristic search algorithm widely used in path finding. It uses a *HeuristicFunction* to estimate the distance between each *HeuristicNode* and the goal. It is complete, and optimal if the heuristic is admissible and consistent. The nodes are visited in a best-first search strategy where the nodes are ordered by their score, namely $f(n) = g(n) + h(n)$.

- **IDA*** [4]: This algorithm is similar to A* but uses iterative deepening to limit the memory usage to the minimum. It uses the score $f(n)$ as the maximum depth bound to cut-off the search. This bound is recomputed iteratively, taking the minimum $f(n)$ score of all of those nodes that exceeded the previous bound.

- **AD*** [7]: It is an advanced state-of-the-art algorithm with replanning capabilities commonly used in path planning. It is able to compute sub-optimal bounded solutions inflating the value of the heuristic, so the cost of the solution can be adjusted depending on the time available to compute it. The solution can be improved iteratively reusing previous computation efforts and managing changes in the costs of the transitions. The order in which the nodes are visited is similar to A*, but taking into account the inflation of the heuristic. This difference requires using a different implementation of *Node*, *ADStarNode*.

Local search:

- **Hill Climbing (HC)** [2]: This algorithm starts the exploration in an arbitrary state and iteratively selects the successor state with the lowest heuristic value (the *HeuristicNode* that is closer to the goal) in order to find a local optimum. The algorithm is neither optimal nor complete, but can provide good solutions in very fast way in some search problems.

- **Enforced Hill Climbing (EHC)** [8]: Is a variation of HC that uses a BFS exploration when the algorithm gets stuck in a local optimum. The algorithm is only complete when the problem has not dead-ends, otherwise it can fail without reaching the goal.

Although these algorithms are enough to solve search problems in a huge variety of fields, it is easy for any user to implement more algorithms based on this model.

IV. CASE STUDY I: OPTIMALWEB SERVICE COMPOSITION

Web services are network-accessible software components whose functional features are mainly defined by the inputs that consume and the outputs they produce. One of the advantages of Web services is to enable greater and easier integration and interoperability among systems through Web service

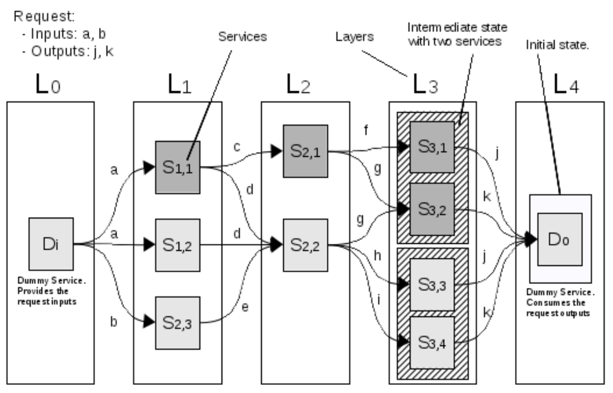

Figure 3. A graph example with 6 layers two different compositions of different sizes.

composition. This advantage allows Web services to be composed mainly by connecting their inputs and outputs to create larger composite services reusing the existing ones. Thus, the goal of Web service composition is to construct new services from existing Web services in order to satisfy some goals which cannot be achieved by single Web service.

There are multiple problems related to the automatic composition of Web services that are still under active research. One of the problems that we tackled in our research [11, 12, 13] is the automatic input/output driven composition of semantic Web services, optimizing both the number of services and the total length of the composition. Concretely, the problem consists of finding the optimal service composition using only the information of their inputs and outputs that solves an input/output request. That is, the optimal composition must use some of the inputs provided and must provided at least all the outputs expected by the user.

To solve one part of this problem, we used Hipster to develop a backwards A* algorithm that searches for the best composition among all possible combinations of services. Given an input-output request, a service graph with all the relevant services for the request is dynamically generated. Then, the backwards A* search algorithm is used to find the minimal service composition that satisfies the request, from the goal outputs to the initial inputs. We also developed different optimizations to reduce the graph size and to dynamically compact functional equivalent nodes to further improve the search speed.

Figure 3 shows an example of a service graph with one optimal composition which consists of 4 services ($S_{1,1}$, $S_{2,1}$, $S_{3,1}$ and $S_{3,2}$). Services are connected by their inputs and outputs, generating longer compositions. In order to find the optimal composition in the graph, the search navigates state by state from the last layer to the first layer, selecting *N* services at each layer. This works as follows: The first state contains only the dummy service D_o, whose inputs are the goal outputs of the request (*j,k*). When the algorithm expands the initial state, it generates two new possible states. Each successor state consists of those services from the previous layer that provide j and k. In this example there are two different successors that satisfy them, one contains the services $S_{3,1}$ and $S_{3,2}$ and the other the services $S_{3,3}$ and $S_{3,4}$. Both successor states have the same cost (2 services) and are located at the same distance to the goal (3 layers), so in the next step the algorithm selects any of them (for example, the first one) and expands it. Now, the successor states are the combination of those services from the previous layer that satisfy the inputs of the current state, which are *f* and *g*. There is only one state that contains one service ($S_{2,1}$) that covers both. This is repeated, using the cost (number of services) and the heuristic (number of layers to the goal) to decide which is state expanded next, until the service that contains the goal state D_i is selected.

From the point of view of the search problem, there are different problem-specific components that we need to implement to perform the backwards A* over the graph, namely: 1) the search states that represent the services selected in each step; 2) the transition function, which is the function that computes the possible successors of each state ; 3) the cost function that calculates the cost of each state and 4) the heuristic function that computes the distance from one state to the goal state.

- **Search states**. For this case, we implemented a class that contains a set of services, the layer index, and some helper methods. The only thing that the state must guarantee is that two state instances with the same services at the same layer must be the same state. To do this, we have to override the methods from the Object Java class *equals* and *hashcode* accordingly. Otherwise, the algorithm cannot differentiate between successors states that are the same and hence the search is turned into a tree search instead of a graph search.
- **Transition function**. This function returns the set of the successor states for a concrete state. This was done by implementing the *TransitionFunction* interface. The function computes the minimum set of all possible combinations of services from the previous layer that provide all the unresolved inputs of the current state (that is, the union of the inputs of each service in the current state).
- **Cost function**. We used the *CostFunction* interface to implement our strategy. The implementation is straightforward: it takes a state as input and returns the number of services that are in the state as output. Hipster has standard implementations for the informed search algorithms to work with double cost types. Since we used doubles to compute the cost, we did not need to implement anything else to work with the default implementation of the A*.
- **Heuristic function**. We created a heuristic strategy that implements the *HeuristicFunction* interface. This function computes an admissible and consistent heuristic, which is the number of layers to the goal, and returns a double value. Again, as we use the default cost type, we do not need anything else. The default implementation knows how to aggregate the cost and the heuristic to guide the search towards the goal.

All these elements (plus the initial state) are provided to the A*, which is then instantiated as an iterator. By iterating over the A* iterator, we can obtain the next expanded node until the current node contains our goal state. An advantage of the iterative model is that we can keep running the algorithm in order to find more than one composition in ascending order according to its size in number of services and length. Results obtained in this work proved the efficiency of the library.

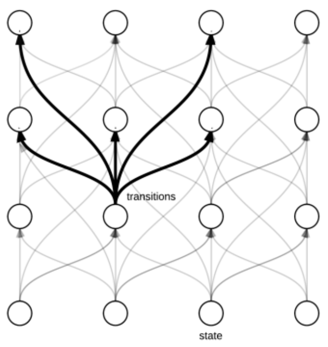

Figure 4. Regular arrangement of the states using the state lattice strategy. The actions connecting the states (in black) are position-independent so they can be used to connect the equally arranged states.

V. CASE STUDY II: MOTION PLANNING

Autonomous vehicles rely on a motion planner to determine the sequence of actions to reach one or more objectives from a starting position. Discretizing the state space of the vehicle has proved to be a successful approach to reduce the computational complexity of the problem. The state lattice is a regular sampling strategy that introduces important benefits: it allows working with a set of actions extracted from the vehicle motion model to connect the discrete states, and because of the regularity these actions are position-independent, so they can be replicated for every pair of states equally arranged (Figure 4). The discrete states and the actions connecting them are expressed as a directed graph, so it is straightforward to obtain the optimal path using a search algorithm. As the motions between states are extracted from the motion model, the path returned by the algorithm is guaranteed to fulfill the maneuvering restrictions of the vehicle.

The planners described in [14, 15] were implemented using the forward implementation of Anytime Dynamic A* (AD*) included in Hipster. AD* has proved its efficiency in the motion planning field, and manages replanning and the obtention of sub-optimal bounded solutions anytime; these solutions can be improved iteratively reusing previous computation effort. Several components of Hipster were implemented taking into account the problem-specific constraints:

- **States**: Are defined by a 5-tuple that contains the vehicle pose and the linear and angular speeds:

$$x = (x_x, x_y, x_\theta, x_v, x_\omega)^T \quad (1)$$

- **Transitions:** The problem requires to store the action used to connect each pair of states, the followed path and the predicted uncertainty along it. To do this we use a custom implementation of the *Transition* object provided with the library, and our *TransitionFunction* implementation returns objects of our custom-defined transition type. Extending these components in Hipster does neither affect the definition of *Node* used by AD* nor the algorithm itself.

- **Cost function**: We evaluate each transition to obtain the probability of avoiding collisions, the time length of the action and the uncertainty at the final state. These measures are stored in a custom-defined cost element. We use an implementation of *CostFunction* that receives objects of our custom transition type and returns this type of cost elements. Most search libraries do not allow custom definitions for the cost of the paths and they assume a numeric value, which limits their usability in cases like this, where complex evaluations cannot be summarized numerically. Nevertheless, one of the benefits of using Hipster is the possibility of using a custom-defined value type, defining the operations of addition —to accumulate the cost of the path from the starting state—, and scaling —to inflate the heuristic value and obtain sub-optimal solutions anytime.

- **Heuristic function**: As heuristic we used the cost of the path without taking into account the vehicle motion model. It is obtained executing a 2D search with the Dijkstra search algorithm implementation provided in Hipster. The search needs to be performed backwards, and it uses a custom-defined stop condition: it explores 1.5 times the cost of the optimal path between the goal and the beginning position. As Hipster algorithms are implemented in a iterative way, which grants total execution control, we could vary the stop condition of the algorithm as detailed in Algorithm 1. This heuristic only depends on the environment so it is executed at the beginning of the planning process. After executing this search the closed queue of the algorithm contains the cost of the 2D path between all the explored positions and the goal. As Hipster provides full access to the internals of the algorithm, accessing these values does not require additional operations For this search our *State* objects are 2D positions. The *TransitionFunction* returns the 8-connected positions neighbors and the *CostFunction* returns the euclidean distance between states. This search problem did not require neither a custom definition for the cost nor transitions, so using Hipster with the default components was straightforward. The *HeuristicFunction* component of the motion planner executes the described 2D Dijkstra search in its initialization.

The library design keeps separated the implementation of the operators from the algorithm, isolating the problem-specific constraints and resulting in a simpler yet efficient planner.

Algorithm 1 Example of custom stop condition using Hipster

```
1: $stop = \infty$
2: $node = end$
3: while $node.cost() < stop$ & $dijkstra.hasNext()$ do
4:     $node = dijkstra.next()$
5:     if $node.state == initial.state$ then
6:         $stop = 1.5 * node.cost()$
7:     end if
8: end while
```

VI. CONCLUSIONS

In this paper we presented Hipster, a heuristic search library for Java. The main goal of Hipster is to provide robust and flexible implementations of the most widely used uninformed and informed search algorithms. Hipster relies on a common data model that is shared among all the implementations, so it is easy to reuse, extend and understand. It comes with implementations of common search algorithms such as Dijkstra or A*, but also with more advanced techniques that are not usually implemented in open source libraries, like the AD* algorithm for path planning. Hipster is licensed under a permissive open source Apache 2.0 license to facilitate the integration in any type of educational, commercial and research projects. The library was successfully integrated in two representative research projects with different requirements and problem-specific operators. As future work, we plan to extend the library with other algorithms such as ARA*, D* or bidirectional search, but keeping the same simple model.

REFERENCES

[1] S. Edelkamp and S. Schrödl, *Heuristic Search: Theory and Applications*. Elsevier, 2012.

[2] S. J. Russell and P. Norving, *Artificial Intelligence: A Modern Approach*. Pearson Education, 2010.

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

[4] R. E. Korf, "Depth-first iterative-deepening: An optimal admissible tree search", in *Artificial intelligence* vol. 27, no. 1, pp. 97-109, 1985.

[5] A. Stentz, "Optimal and efficient path planning for partially-known environments", in *IEEE International Conference on Robotics and Automation (ICRA)*, 1994, pp. 3310-3317.

[6] M. Likhachev, G. J. Gordon and S. Thrun, "ARA*: Anytime A* with Provable Bounds on Sub-Optimality", in *Advances of Neural Information Processing Systems 16 (NIPS)*, 2003.

[7] M. Likhachev, D. Ferguson, G. J. Gordon, A. Stentz and S. Thrun, “Anytime Dynamic A*: An anytime, replanning algorithm”, in *International Conference on Automated Planning and Scheduling (ICAPS)*, 2005, pp. 262-271.

[8] J. Hoffman, “FF: the fast-forward planning system”, in *Artificial Intelligence magazine*, vol. 22, no. 3, pp. 57, 2001.

[9] F. Glover, “Future paths for integer programming and links to artificial intelligence”, in *Computers and Operations Research*, vol. 13, no. 5, pp. 533-549, 1986.

[10] S. Kirkpatrick, C. D. Gelatt, M and P. Vecchi, “Optimization by Simulated Annealing”, in *Science,* vol. 220, no. 4598, pp. 671-680, 1983.

[11] P. Rodriguez-Mier, M. Mucientes, J.C. Vidal and M. Lama, “An Optimal and Complete Algorithm for Automatic Web Service Composition”, in *International Journal of Web Services Research (IJWSR)*, vol. 9, no. 2, pp. 1-20, 2012.

[12] P. Rodriguez-Mier, M. Mucientes and M. Lama, “Automatic Web Service Composition with a Heuristic-Based Search Algorithm”, in *IEEE International Conference on Web Services (ICWS)*, pp. 81-88, 2011.

[13] P. Rodriguez-Mier, M. Mucientes and M. Lama, “A Dynamic QoS-Aware Semantic Web Service Composition Algorithm”, in *International Conference on Service Oriented Computing (ICSOC)*, 2012.

[14] A. González-Sieira, M. Mucientes and A. Bugarín, “Anytime Motion Replanning in State Lattices for Wheeled Robots”, in XIII Workshop of Physical Agents, pp. 217-224, 2012.

[15] A. González-Sieira, M. Mucientes and A. Bugarín, “A State Lattice Approach for Motion Planning under Control and Sensor Uncertainty”, in *First Iberian Robotics Conference (ROBOT)*, pp. 247-260, 2013.