Weighted Randomized Anytime Planning in Pyhop

Gabriel J. Ferrer

Hendrix College
1600 Washington Ave.
Conway, Arkansas 72034 USA
ferrer@hendrix.edu

Abstract

Since they produce plans whose quality increases with time, anytime planners are very useful for domains such as robotics and video games. Planners using the SHOP algorithm can operate as anytime planners by retaining the result of the first depth-first search path that reaches the goal, then returning the results of subsequent searches if they improve upon it. However, backtracking to the most recent alternative may not be the best approach for quickly finding a low-cost plan.

In this paper, we replace backtracking depth-first search with a randomized algorithm in the Pyhop implementation of SHOP. Whenever there are multiple options, the planner selects a random operator or method. For every selected option, it records the cost of every plan that includes that option. It uses these cost statistics to make the selection of options that lead to lower-cost plans more probable.

We evaluated the resulting HTN planner on three domains - the Traveling Salesperson Problem, the Pickup and Delivery Problem, and the Satellite Problem. Our experiments show that the weighted-selection approach outperforms both depth-first search and unweighted randomized selection.

Introduction

Anytime planners (Dean and Boddy 1988) produce plans with quality proportionate to available planning time. They are very useful for domains with real-time constraints such as robotics and video games. The Simple Hierarchical Ordered Planner (SHOP) (Nau et al. 1999) was introduced to enable implementing Hierarchical Task Network (HTN) planners for many different domains. The Pyhop implementation of SHOP (Nau 2013) enables an HTN planner to work with arbitrary data structures in the Python language.

We have built an anytime planner atop Pyhop called **pyhop-anytime** (Ferrer 2024d). Within **pyhop-anytime** we have implemented three anytime planners in the `Planner` class in the file `pyhop.py`:

- The time-limit feature described in Section 3.2.4 of the SHOP3 manual (Goldman and Nau 2019), implemented in the `anyhop()` method of the `Planner` class.
- A variant of SHOP in which operators and methods are chosen randomly whenever there is more than one option. There is no backtracking; instead, a new randomized depth-first search is launched whenever the previous search finds a plan or fails. Each plan is generated by the `randhop()` method, called repeatedly by the `anyhop_random()` method until time runs out.
- Another randomized planner, the **action tracker**, in which operators and methods that lead to higher-quality plans have a higher probability of being selected. Each plan is generated by the `make_action_tracked_plan()` method, called repeatedly by the `anyhop_random_tracked()` method until time runs out.

Our anytime planners are designed for domains in which finding an **optimal-cost plan** is **NP-Complete** but finding **non-optimal valid plans** is **feasible in polynomial time**. The planners employ whichever cost function is supplied for a given domain. We experimentally evaluated these planners in the following domains:

- The Traveling Salesperson Problem (TSP).
- The Pickup and Delivery Problem (PDP) is built atop the same graph structure as the TSP, but augmented with packages that must be transported between nodes.
- The STRIPS Satellite domain (sat 2002) from the 2002 International Planning Competition (ipc 2002) provides a number of standardized problems for a traditional STRIPS planning domain.

In nearly all experiments, the purely random planner outperformed the backtracking planner. The action tracking planner sometimes was tied with the backtracking planner but almost always significantly outperformed both of them.

After an overview of the SHOP planning algorithm, we describe our randomized variants, our experiments, and our results, followed by conclusions and future work.

The SHOP Algorithm

In the SHOP planning algorithm (Algorithm 1)[1], **operators** specify state transformations and **methods** decompose tasks into lists of subtasks, in some cases with multiple alternatives (Nau et al. 1999). The subtasks themselves consist of methods or operators. A **plan** is a sequence of operators that completes the given tasks from the given starting state.

Many planning domains are notorious for their intractable computational complexity (Bylander 1994) (Gupta and Nau

[1]Methods `anyhop()` and `pyhop_generator()` of our `Planner` class and `successors()` of our `PlanStep` class.

Algorithm 1: The SHOP algorithm

Input: state, tasks, plan, operators, methods
Output: plan

```
1: if No tasks remaining then
2:     return plan
3: else if First task in operators then
4:     Create newstate by applying operator to state
5:     return shop(newstate, remaining tasks, plan with op-
       erator, operators, methods)
6: else if First task in methods then
7:     for Every method relevant to the first task do
8:         if The method has relevant subtasks then
9:             return shop(state, subtasks + remaining tasks,
               plan, operators, methods)
10:        end if
11:    end for
12: else
13:    return failure
14: end if
```

1992). The SHOP algorithm is well-suited for domains in which finding an optimal-cost plan is NP-Complete but finding non-optimal valid plans is feasible in polynomial time. In these domains, we find a valid plan in polynomial time without backtracking by making arbitrary decisions when nondeterministic choices are encountered[2]. Each backtrack to a nondeterministic choice might produce an improved plan. When a time limit is reached, the best plan found is returned. Increased time limits create more opportunities to find plans that are closer to the optimal cost.

One such domain is the Pickup and Delivery Problem (PDP), which has been heavily studied in the planning literature (Coltin 2014). Each PDP instance consists of an undirected weighted graph, a list of packages to deliver (including origins and destinations), and a robot with a specified carrying capacity. The SHOP methods below[3] find a correct plan in polynomial time without any backtracking. As a preprocessing step, we run the Floyd-Warshall algorithm for finding the shortest paths between all pairs of nodes.

- **deliver-all-packages-from**:
  - If all packages are at their destinations, return success.
  - Otherwise, create a list of pairs of packages and destinations that includes every package aboard the robot along with its destination. If the robot has spare capacity, include every undelivered package with its starting location. Post **possible-destinations** with this list.
- **possible-destinations**: Examine each destination given by the provided destination list.
  - If any destination matches its current location, post **pick-up** or **put-down** depending on the destination's purpose. Then post **deliver-all-packages-from**.
  - If not, nondeterministically choose one neighbor which is part of a shortest path from the current location to a destination in the list. Post **progress-task-list** with that neighbor and a list of only those destinations whose shortest paths include the chosen neighbor.
- **progress-task-list**: Add a **move-one-step** operator to the chosen neighbor. Invoke **possible-destinations** with all of the destinations remaining in its list.

There are three operators in this domain:

- **pick-up**: Pick up an object at the current node.
- **put-down**: Put down an object at the current node.
- **move-one-step**: Move to a neighboring node.

When combined with the SHOP algorithm, the above methods produce a planner that guarantees the delivery of every package. The combination of **possible-destinations** and **progress-task-list** on each invocation will move a package closer to its destination in one of four ways:

1. It picks up a package from its starting location.
2. It puts down a package at its destination.
3. It moves one step closer along the shortest possible path to the destination of a package it has already picked up.
4. It moves one step closer along the shortest possible path to the origin of a package it needs to pick up.

Once a delivery takes place, **deliver-all-packages-from** generates a list of possible pickups and deliveries that includes a strictly smaller number of packages than its previous invocation. Thus, the combination of these three methods is guaranteed to eventually deliver all of the packages, and as every **move-one-step** operator always moves along a shortest path, the overall length of the plan is bounded above by the longest shortest path plus two, multiplied by the number of items to deliver. This is therefore an example of producing a polynomial-time algorithm for finding valid (but not necessarily optimal-cost) plans by combining the SHOP algorithm with suitable domain-specific methods.

Randomizing the SHOP Algorithm

Simple Randomized SHOP

We define an **option** as one possible operator or method that the planner can select. Simple Randomized SHOP[4] is identical to Algorithm 1, except that whenever a method chooses an option nondeterministically (e.g., **possible-destinations**) or multiple methods are options (line 7 of Algorithm 1), it selects one option at random and continues planning without backtracking. After it completes a plan, it keeps generating plans until it reaches its time limit[5].

Weighted Randomized SHOP

Weighted Randomized SHOP[6] is similar to Simple Randomized SHOP, except that instead of selecting options with uniform probability we instead select them according to a dis-

[2]Domains for which the SHOP algorithm might fail to find a plan without backtracking are outside the scope of this paper.

[3]Implemented in `graph_package_world.py` in the `pyhop_anytime_examples` directory

[4]Method `randhop()` of our `Planner` class

[5]Method `anyhop_random()` of our `Planner` class

[6]Method `make_action_tracked_plan()` of `Planner`.

tribution informed by their utility. As with Simple Randomized SHOP, we invoke Weighted Randomized SHOP to repeatedly generate plans until it reaches its time limit[7].

We construct this distribution as follows. For every option we record the following information[8] for every randomly generated plan that employed that option at some point:

- Total cost and number of all successful plans
- Maximum cost of any successful plan
- Number of failed planning attempts

Using the above information, we define a total ordering on options[9] by assigning higher priority to whichever option has lower mean cost[10].

Whenever multiple options have cost information available, each option has a probability assigned as follows[11]:

- The overall set of options is divided into two groups: those that have been selected for a plan and those that have not. Each group is given a probability budget based on the proportion of options in that category.
- Each unselected option has the same probability of selection, receiving an equal share of the unselected budget.
- Those that have been selected are sorted according to the priority scheme outlined above. Each option is assigned a probability double that of the option of next lowest priority, yielding exponentially decreasing probabilities. The sum of these probabilities is the selected-option budget.
- By using a ranking system, we determine selection probabilities independently of the domain's cost function.

We expect that options initially included in plans of low average cost will be included in additional low-cost plans. If this proves to be the case, these options will continue to be selected frequently. If not, these options will diminish in the ranking and be selected less often. Options that initially led to higher cost plans are then more likely to be selected, and if further exploration demonstrates their utility they will be selected more frequently later on.

By default, the tracker does not record outcomes if a given option is the only option. We created a variant[12] in which we record outcomes for every option, even if a given method or operator is the only choice at that point.

Related Work

Another randomized version of **pyhop** is **pyhop-m** (Shao et al. 2021). It expands upon **pyhop-h** (Cheng et al. 2018), an implementation of Pyhop incorporating heuristic search in place of depth-first search. **pyhop-m** replaces **pyhop-h**'s heuristic using the costs of a number of random plans to estimate the quality of each alternative.

[7]Method `anyhop_random_tracked()` of `Planner`.

[8]Recorded in an object of `ActionTracker`.

[9]See our `OutcomeCounter` class.

[10]Although it is beyond the scope of this paper, our implementation also takes plan failures into account by incorporating the number of plan failures into the total ordering.

[11]Method `distribution_for()` of `ActionTracker`.

[12]Set `ignore_single` to `False`.

Our work differs from **pyhop-m** in two critical ways. First, our formulation of anytime planning relies upon a depth-first search implementation of SHOP. As **pyhop-m** employs randomization to serve as an estimator for a heuristic search implementation of SHOP, it is not at all clear to us how it could be converted into an anytime algorithm.

Second, since **pyhop-m** uses random plans to construct a heuristic estimate of distance to a goal, it discards those plans once that estimate has been calculated. In contrast, we employ random plans to both update the probability distribution for operator and method selection that guides our search algorithm as well as to serve as candidate solution plans.

Experimental Analysis

Setup

We evaluated four **pyhop-anytime** variations:

1. **DFS**: Depth-first **pyhop** with a time limit.
2. **Random**: Simple randomized **pyhop**.
3. **Tracker1**: Weighted randomized **pyhop**, tracking all except single-alternative options.
4. **Tracker2**: **Tracker1** modified to track all options.

We tested these four variations in these domains:

1. Traveling Salesperson Problem (TSP)
2. Pickup and Delivery Problem (PDP) with a single robot
3. Satellite coordination problem

TSP shows the performance of our approach in a domain in which every operator selection requires a nondeterministic choice. PDP shows performance in a domain akin to TSP but more similar to a typical HTN planning problem. Satellite shows performance on a well-known benchmark STRIPS planning domain.

Our TSP implementation has one operator (**move** - moves from one city to another) and one method (**complete-tour-from**). The method arbitrarily selects a previously unvisited node to be the next node visited, and then invokes itself recursively. Our PDP implementation is described in the SHOP Algorithm section.

To generate TSP instances, we randomly generate (x, y) coordinates for each city on a 100x100 grid. We then assign edge weights between every pair of cities using the Euclidean distance.

To generate PDP instances, we randomly generate (x, y) coordinates for 36 nodes on a 100x100 grid. For each pair of nodes, there is a 25% chance that we insert an edge. As with TSP, the edge weight is determined by the Euclidean distance. Each of 12 packages are assigned to a distinct random starting node and are assigned a distinct random goal node. The robot can carry up to three packages.

Results and Analysis

We ran four initial experiments (Ferrer 2024b). TSP results are in Figures 1, 2, and 3 and PDP results are in Figure 4. TSP experiments used 15, 30, and 50 cities, with 5, 10, and 30 second limits respectively. PDP used a 30 second limit.

The cost of a TSP plan is the sum of the weights of each edge traversed by the **move** operator. The cost of a PDP

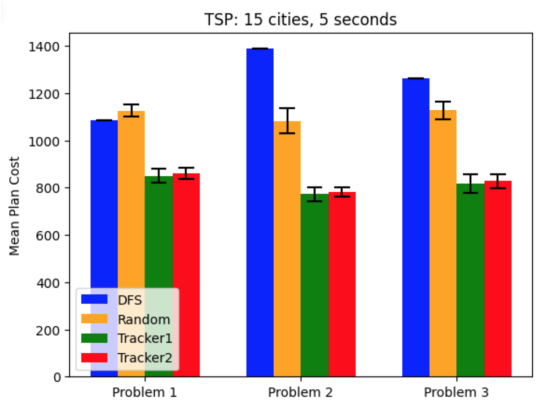

Figure 1: TSP: 15 Cities, 5 seconds

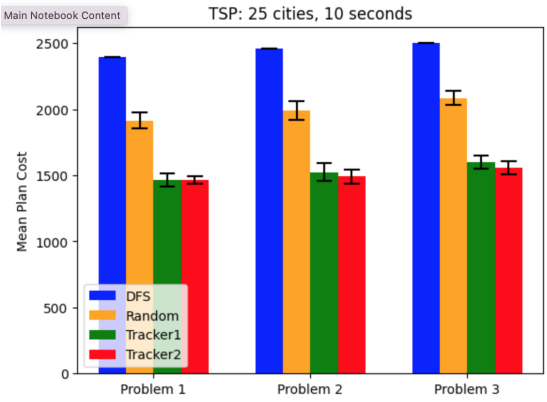

Figure 2: TSP: 25 Cities, 10 seconds

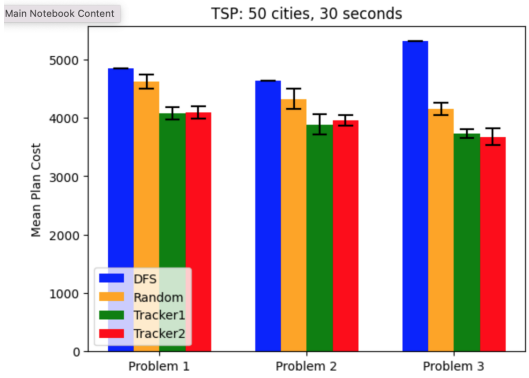

Figure 3: TSP: 50 Cities, 30 seconds

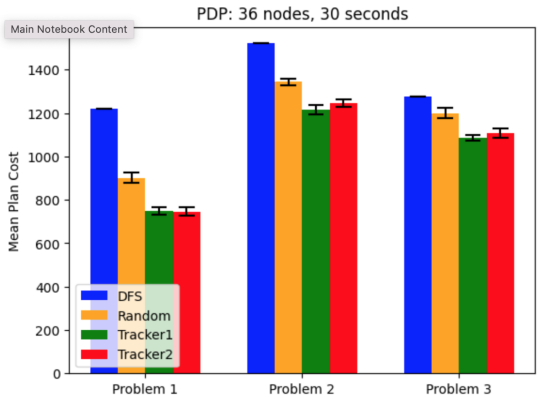

Figure 4: Pickup/Delivery Problem, 36 nodes, 30 seconds

plan is the sum of the weights of each edge traversed by the **move-one-step** operator, with each **pick-up** and **put-down** operator having a cost of 1.

We randomly generated three problem instances for each experiment. For each problem instance, we ran the deterministic DFS planner once and each randomized planner nine times. For the randomized planners, we report the mean plan cost across the nine runs for each problem instance, with error bars indicating a 95% confidence interval.

In the smallest TSP problems (see Figure 1), **Random** typically outperformed **DFS**, but for one problem **DFS** was superior. In all cases, **Tracker1** and **Tracker2** outperformed both **DFS** and **Random** to a degree that their 95% confidence intervals never overlapped. The difference between **DFS**, **Random**, and the **Tracker** implementations diminished on the largest (50 cities) TSP problems as well as the PDP problems but remained statistically significant.

This result seems intuitive. As problem size increases, plan cost as well as the number of distinct operator and method options also increases beyond the degree to which we increased the time budget. Consequently, less information is available for each option, and our probability distribution is less informed. The information is still valuable, but the impact is not as drastic as with smaller problem sizes.

To investigate the impact of increasing the time budget, we ran two additional experiments (Ferrer 2024c) of 200 second duration on the 50 city TSP problem (Figure 5) as well as the PDP problem (Figure 6). On the TSP problem, this replicated the ratio between the costs of the action-tracked plans and the fully randomized plans on the 25 city problem. On the PDP problem, the additional time improved the ratio only slightly. The relative simplicity of the TSP domain seems to make it more amenable to performance improvements from increasing the time budget.

For the Satellite domain, we used the five largest problems in the repository (Ferrer 2024a). In Figure 7, we see that DFS often performed well - in those situations, action tracking matched its performance. In other situations, action tracking greatly outperformed DFS.

Tracker1 was our initial implementation, as we did not find it intuitive to track options that the planner was required to include. **Tracker2** tested that intuition. As they show statistically indistinguishable performance in all experiments, this design decision seems to have no impact at all.

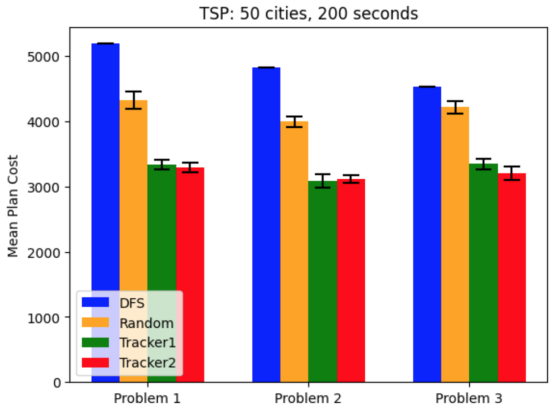

Figure 5: TSP: 50 Cities, 200 seconds

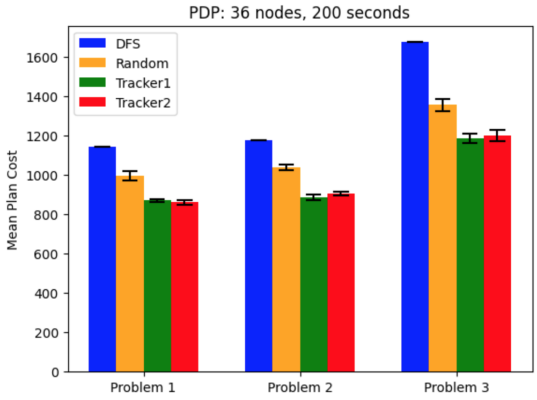

Figure 6: Pickup/Delivery Problem, 36 nodes, 200 seconds

Conclusion

Performance of anytime planning with the SHOP algorithm usually improves by replacing backtracking with randomization. Randomization with operator and method tracking produces significantly larger and more consistent improvements. Future work includes investigating changes to the tracking scheme to enable further improvements with limited time budgets, assessing performance in a wider variety of planning domains, incorporating **pyhop-anytime** into a planning and execution system for the robots in our lab, and potentially incorporating **pyhop-anytime** (Ferrer 2024d) into the **GTPyhop** planner (Nau 2021).

References

2002. International Planning Competition. https://ipc02.icaps-conference.org/.

2002. Satellite STRIPS Domain. https://ipc02.icaps-conference.org/CompoDomains/SatelliteStrips.pddl. Accessed: 2024-05-20.

Bylander, T. 1994. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69(1-2): 165–204.

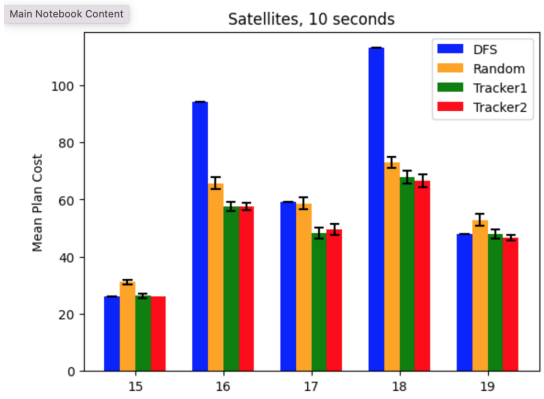

Figure 7: Satellite Domain, 10 seconds

Cheng, K.; Wu, L.; Yi, X. H.; Yin, C. X.; and Kang, R. Z. 2018. Improving Hierarchical Task Network Planning Performance by the Use of Domain-Independent Heuristic Search. *Knowledge-Based Systems*, 142: 117–126.

Coltin, B. 2014. *Multi-Agent Pickup and Delivery Planning with Transfers*. Ph.D. thesis.

Dean, T.; and Boddy, M. 1988. An Analysis of Time-Dependent Planning. In *Proceedings of the 7th National Conference on Artificial Intelligence*.

Ferrer, G. 2024a. Anyhop Satellite Experiments. https://www.kaggle.com/code/gabrielferrer/anyhop-satellite-experiments. Accessed: 2024-05-20.

Ferrer, G. 2024b. Bar Plots for ICAPS-HPlan 2024 paper. https://www.kaggle.com/code/gabrielferrer/bar-plots-for-icaps-hplan-2024-paper. Acccessed: 2024-04-24.

Ferrer, G. 2024c. Extended Experiments for ICAPS-HPlan 2024 paper. https://www.kaggle.com/code/gabrielferrer/extended-experiments-for-icaps-hplan-2024-paper. Accessed: 2024-04-24.

Ferrer, G. 2024d. pyhop_anytime. https://github.com/gjf2a/pyhop_anytime. Accessed: 2024-04-24.

Goldman, R. P.; and Nau, D. 2019. SHOP3 Manual. https://shop-planner.github.io. Accessed: 2024-03-19.

Gupta, N.; and Nau, D. 1992. On the Complexity of Blocks-World Planning. *Artificial Intelligence*, 56(2-3): 223–254.

Nau, D. 2013. Pyhop. https://bitbucket.org/dananau/pyhop/src/master/. Accessed: 2024-03-19.

Nau, D. 2021. GTPyhop. https://github.com/dananau/GTPyhop. Accessed: 2024-03-19.

Nau, D. S.; Cao, Y.; Lotem, A.; and Muñoz-Avila, H. 1999. SHOP: Simple hierarchical ordered planner. In *Proceedings of the 1999 International Joint Conference on Artificial Intelligence (IJCAI)*, 968–973.

Shao, T.; Zhang, H.; Cheng, K.; Zhang, K.; and Bie, L. 2021. The Hierarchical Task Planning Method Based on Monte Carlo Tree Search. *Knowledge-Based Systems*, 225.