An Exploration of Novel Beam Search Variants

Bryan McKenney

Department of Computer Science
University of New Hampshire, USA
Bryan.McKenney@unh.edu

Abstract

Anytime heuristic search algorithms, which find a poor solution quickly and better ones over time, are useful when there is an unknown amount of time to solve a planning problem. Rectangle search (Lemons et al. 2024) is a state-of-the-art anytime algorithm based on beam search, but it has some drawbacks — a parameter that is hard to set and poor performance in some planning domains. In this paper, I explore the performance of three novel beam search variants, two of which are anytime algorithms inspired by rectangle search, and show that one of them, outstanding search, has potential as an alternative to rectangle search.

Introduction

Anytime heuristic search algorithms are useful when there is an unknown amount of time to solve a problem. Anytime algorithms quickly find a suboptimal solution and then find increasingly better solutions over time, possibly until the optimal solution is found. Rectangle search (Lemons et al. 2024) is an anytime heuristic search algorithm that is based on beam search (Newell 1978), which is breadth-first search with a limited width. Beam search is not anytime, nor is it complete, but rectangle search is both. It keeps an open list for each depth level and expands one node from each existing level at each iteration i and then i nodes from a number of new depth levels equal to the *aspect ratio* parameter. Rectangle search is competitive with beam search in its first solution quality and with other anytime algorithms, such as ARA* (Likhachev, Gordon, and Thrun 2003), in its anytime performance. In many domains, using a large aspect ratio such as 500 will result in better performance than a small aspect ratio such as 1. However, there are some domains for which even this trick does not help. An example is given in Figure 1, where rectangle search with an aspect ratio of 1 and of 500 both do very poorly compared to ARA* and CABS in the structured grid pathfinding problem 64room (Sturtevant 2012). It is not clear how to set the aspect ratio to improve performance in this domain, or if any setting will lead to good performance.

Wheeler Ruml proposed a more flexible anytime search algorithm called *outstanding search* that has some similarities to rectangle search and incorporates ideas from the original beam search (Newell 1978), which has a variable-width beam, and limited discrepancy search (Harvey and Ginsberg 1995), which assumes that the heuristic is equally inaccurate everywhere in the search space and tries to account for that. Outstanding search, like rectangle, keeps an open list for each depth level, but, unlike rectangle, it can compare nodes across depth levels and be more selective about what to expand next, and it also does not have an aspect ratio parameter.

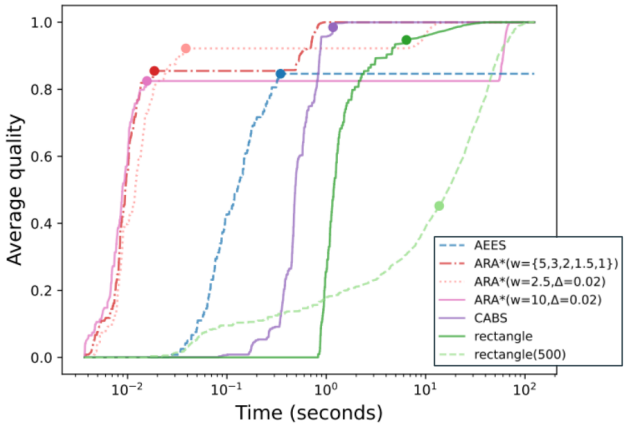


Figure 1: The anytime performance of various algorithms, including rectangle search, on handcrafted grid pathfinding problems. The dot indicates when a solution has been found for every instance. Plot from Lemons et al. (2024).

In this paper, I describe three novel beam search variants — threshold bead search, outstanding search, and outstanding rectangle search — and compare them to the state-of-the-art algorithms bead search (Lemons et al. 2022) and rectangle search on several planning domains. Threshold bead search and outstanding rectangle search do not appear promising, but outstanding search has some potential as a more consistent algorithm than rectangle search that finds decent first solutions.

Previous Work

In this section, I describe the algorithms beam search, bead search, rectangle search, and limited discrepancy search.

Beam Search

Beam search was invented by Newell (1978). It aims to find a single solution quickly but has no guarantee that it will find a solution even when one exists (i.e. it is not a complete algorithm). Beam search works by selecting a set of nodes to expand at each depth and throwing out the rest. Modern beam search uses a beam width parameter b to specify the number of nodes that will be kept at each depth; it expands the b nodes with the lowest cost-to-go estimates h. Newell's beam search is quite different. Instead of a parameter to fix the width of the beam, it has a threshold parameter e. Any node at a new depth with score x that satisfies $|x - x^*| \leq e$, where x^* is the best score of any node at that depth, will be retained in the beam. This means that the beam width can fluctuate across depth levels depending on how many promising nodes there appear to be at each level. The "score" mentioned is just some heuristic measure of how promising a node is; Newell uses the likelihood that a state would follow the previous state.

Bead Search

Bead search was introduced by Lemons et al. (2022). It is simply modern beam search (fixed-width beam) that uses the distance-to-go estimates d instead of the cost-to-go estimates h to determine which b nodes will be expanded at the next depth. In practice, this leads to finding a solution faster.

Rectangle Search

Rectangle search is a complete anytime algorithm developed by Lemons et al. (2024). Unlike beam search, which shoots through the search space once and never returns to a depth it has previously expanded nodes at, rectangle search returns to the shallowest depth in each iteration and gradually increases its "beam width" until it explores the entire search space. It keeps an open list for each depth level, and at each iteration i it expands 1 node from each existing depth and then i nodes from $aspectRatio$ new depths, where $aspectRatio$ is a parameter of the algorithm. This means that the algorithm explores an ever-expanding square (when $aspectRatio = 1$) or rectangular portion of the search space. Aspect ratios of 1 and 500 are compared by Lemons et al. (2024), and each is good in different domains, although rectangle-500 seems to be better overall.

While rectangle search has state-of-the-art performance in most domains, there are some where it is sub-par (as shown in Figure 1), including grid pathfinding on handcrafted or structured (as opposed to completely randomly generated) grids.

Limited Discrepancy Search

Limited discrepancy search (LDS) was invented by Harvey and Ginsberg (1995). It explores paths through the search tree with an increasing number of discrepancies from the "best" path according to the heuristic in order to account for heuristic error. The assumption is that the heuristic is equally inaccurate everywhere in the tree. It is not important to know exactly how LDS works to understand this paper — the idea of discrepancies is adopted for threshold bead and outstanding search, but used in a completely different way.

Approach

In this section, I describe the novel beam search variants explored in this work — threshold bead search, outstanding search, and outstanding rectangle search — and then the planning domains in which they are tested. I implemented these algorithms in C++ in the codebase of Lemons et al. (2024).

Threshold Bead Search

Threshold bead search is Newell (1978)'s beam search but using d as the score function instead of likelihoods. I chose to use d instead of h because of the success of bead search over beam search. At each depth, threshold bead calculates a *discrepancy score* for each node

$$discrepancyScore_n = d_n - d_{best}$$

where d_n is the d-value of node n and d_{best} is the lowest d-value of any node at that depth. It retains all nodes at that depth that satisfy

$$discrepanyScore_n \leq threshold$$

where $threshold$ is a parameter to the algorithm. Note that if all nodes at a depth have the same d value, they will all have discrepancy scores of 0 and thus will all be retained for expansion.

Outstanding Search

Outstanding search was invented by Wheeler Ruml. It uses the same definition of discrepancy scores that threshold bead uses, and it compares nodes across depth levels using their discrepancy scores when deciding which to expand. Outstanding search only expands one promising-looking node from any depth per iteration instead of one node from each depth and many more at new depths. This gives it the flexibility to explore *outstanding* parts of the search space rather than exploring in a strictly rectangular pattern. Like with LDS, the assumption here is that the heuristic (d, in this case) is equally inaccurate everywhere in the search space, which is why discrepancies can be used to compare nodes at different depth levels.

Algorithm 1 outlines outstanding search. Like rectangle search, it utilizes an open list for each depth level (line 1). The smallest d of any node seen at each depth is recorded as well (line 2), and updated whenever a node is added to an open list, as can be seen in Algorithm 2. The smallest d value for an open list is used to calculate a discrepancy score for each node in that open list according to Algorithm 3. Algorithm 4 shows how nodes are selected and expanded. The node with minimal discrepancy score across depths is selected (line 29) and its children are added to the open list at the next depth level (line 47). If there is a tie between nodes of the same depth, the node with lowest f is selected, and, if there is still a tie, the node with lowest h is selected. If there is a tie between nodes of different depths, the node with shallowest depth is selected. A child is pruned if its f is greater than or equal to the incumbent solution cost (line 40).

Algorithm 1: OUTSTANDINGSEARCH(k, $start$)

```
1: openlists ← [∅, ∅]
2: dBestInList ← [∞, ∞]
3: closed ← ∅
4: incumbent ← node with g = ∞
5: nodesExpanded ← 0
6: nodesExpandedAtDeepestDepth ← 0
7: children ← expand(start)
8: for each child in children do
9:     AddToOpenlist(child, 0)
10: CalcDiscrepancyScores(0)
11: depth ← 1
12: while non-empty lists exist in openlists do
13:     SelectAndExpand
14:     if nodesExpandedAtDeepestDepth = k or
        nodesExpanded = k · depth then
15:         CalcDiscrepancyScores(depth)
16:         extend openlists with ∅
17:         extend dBestInList with ∞
18:         increment depth
19:         nodesExpandedAtDeepestDepth ← 0
20: return incumbent
```

Algorithm 2: ADDTOOPENLIST(n, i)

```
21: add n to openlists[i]
22: if d(n) < dBestInList[i] then
23:     dBestInList[i] ← d(n)
24:     return true
25: return false
```

Algorithm 3: CALCDISCREPANCYSCORES(i)

```
26: for each n in openlists[i] do
27:     discrepancyScore(n) ← d(n) − dBestInList[i]
```

Algorithm 4: SELECTANDEXPAND

```
28: repeat
29:     n ← node with min discrepancyScore across
        openlists[0..depth − 1]  // includes tie-breaking
30:     i ← index of n's openlist
31:     remove n from openlists[i]
32: until f(n) < g(incumbent)
33: add n to closed
34: children ← expand(n)
35: increment nodesExpanded
36: if i = depth − 1 then
37:     increment nodesExpandedAtDeepestDepth
38: dBestChanged ← false
39: for each child in children do
40:     if f(child) < g(incumbent) then
41:         if child is a goal then
42:             incumbent ← child
43:             report new incumbent
44:         else
45:             dup ← child's entry in closed
46:             if child not in closed or g(child) < g(dup)
                then
47:                 dBestChanged ← AddToOpen-
                    list(child, i + 1) or dBestChanged
48: if dBestChanged and i + 1 < depth then
49:     CalcDiscrepancyScores(i + 1)
```

Outstanding search has a *cautiousness* parameter k which ensures that enough exploration is done in previous depths before unlocking the next depth level for expansion. The criterion for unlocking the next depth is that k nodes must have been expanded at the deepest unlocked depth or $k \cdot depth$ nodes must have been expanded across all depths (line 14), where $depth$ is the current number of depth levels in the search (excluding the locked depth). There is one exception to the cautiousness rule — if there are no nodes to expand except at the locked depth, that depth is unlocked regardless of the number of nodes that have been expanded so far. Discrepancy scores are first calculated for an open list when that depth is unlocked (line 15) and they must be updated whenever a node with d lower than the previous best is added to the open list (line 49).

Outstanding Rectangle Search

Outstanding rectangle search, as the name implies, is a combination of outstanding search and rectangle search. Specifically, in each iteration i, a node to expand is selected in the same manner as in outstanding search — by the lowest discrepancy score across depth levels. After that node is expanded, 1 node is expanded from all existing lower depths and then i nodes are expanded from $aspectRatio$ new depths, where $aspectRatio$ is a parameter of the algorithm. Note that outstanding rectangle search does not need the cautiousness parameter k because it goes deeper with each iteration. Thus, the only difference between this algorithm and rectangle search is which depth it returns to at the beginning of each iteration: rectangle always returns to depth 1, while outstanding rectangle can go to whichever depth it deems has an outstanding node.

Test Domains

I tested these algorithms against bead and rectangle on the following planning domains (on the same instances used by Lemons et al. (2024), 100 instances per domain):

- **tiles**. The 4x4 sliding-tile puzzle, commonly referred to as the 15-puzzle. **unit**, **inv**, and **heavy** cost models are used.
- **vacuum**. A robot on a 200x200 grid with 4-way movement must clean up 10 dirt tiles. **unit** and **heavy** cost models are used.
- **pancake**. Parts of a stack of 70 pancakes of different sizes must be flipped to turn the stack into a pyramid. **unit** and **heavy** cost models are used.
- **64room**. Grid pathfinding with 8-way movement on a map containing 64 rooms. Map from Sturtevant (2012). **unit** cost model is used.

- **orz100d**. Grid pathfinding with 8-way movement on a map from the video game *Dragon Age: Origins*. Map from Sturtevant (2012). **unit** cost model is used.

Results

Bead was tested with beam widths of 30, 100, 300, and 1000. Threshold bead (thresholdbead) was tested with thresholds of 2, 4, and 6 (except in the vacuum domain, where 30, 100, and 150 were used instead). Rectangle and outstanding rectangle (outstandingrect) were tested with aspect ratios of 1 and 500. Outstanding was tested with cautiousness values of 2, 20, 200, and 2000. All algorithms had 7.5G of memory and 5 minutes to solve each instance.

In this section, I will first present the results of these experiments in terms of anytime performance, and then in terms of the first solution. Then I will show some visualizations of how the anytime algorithms search.

Anytime Performance

The following sets of three plots show the average quality, number of instances solved, and solution cost, respectively, as a function of time in a certain domain with a certain cost model. Average quality is calculated by the current best solution cost for an instance divided by the optimal solution cost for that instance, averaged across all instances. The dot indicates when a solution has been found for all 100 instances. Bead and thresholdbead are omitted from these plots for clarity and because they do not have anytime performance — they will be shown in the first solution section.

In **tiles** (Figure 2, Figure 3, and Figure 4), outstanding is in-between rectangle-1 and rectangle-500. Outstandingrect-1 is slightly worse than rectangle-1 and outstandingrect-500 is much worse than rectangle-500. $k = 2$ is the best choice for outstanding here.

In **vacuum** (Figure 5 and Figure 6), outstanding is much better than rectangle-1 and is similar to rectangle-500. Outstandingrect-1 is about the same as rectangle-1 and outstandingrect-500 is much worse than rectangle-500. $k = 20$ is the best choice for outstanding here.

In **pancake** (Figure 7 and Figure 8), outstanding failed to solve any instances because it ran out of memory (more on this later). Outstanding-500 lines up with outstanding-1 and rectangle-1 here and they are all much worse than rectangle-500.

In the grid pathfinding domains **64room** (Figure 9) and **orz100d** (Figure 10), outstanding-2 is between rectangle-1 and rectangle-500 (although in 64room, it finds all solutions faster than rectangle-500). Outstandingrect and rectangle are closely matched (although in 64room, outstandingrect-1 outperforms rectangle-1). $k = 2$ is the best choice for outstanding here.

Overall, outstandingrect is a worse version of rectangle and outstanding has performance in-between rectangle-1 and rectangle-500 (besides on pancake instances).

First Solution

The following sets of plots show the first solution cost and time for the algorithms in a certain domain with different cost models. Each point is a different parameter value. If an algorithm with a certain parameter value could not solve all instances, the point is not plotted.

In **tiles** (Figure 11), bead is the best and thresholdbead is a close second. Outstanding is competitive with rectangle and outstandingrect is bad.

In **vacuum** (Figure 12), thresholdbead and outstandingrect are bad. Outstanding is decent. There is no clear best algorithm here.

In **pancake** (Figure 13), thresholdbead and outstanding failed to solve all instances with all parameter values because they ran out of memory. The same is true for outstandingrect with one of its parameter values.

In **64room** (Figure 14 left), outstanding is the best, outstandingrect is second, and thresholdbead is bad. In **orz100d** (Figure 14 right), rectangle is the best, outstanding is second, and outstandingrect and thresholdbead do not do well.

Overall, outstandingrect is again a worse version of rectangle, outstanding has decent performance (besides on pancake instances), and thresholdbead is not good.

Visualization

Figure 15 shows visualizations of the nodes that outstanding-2, rectangle-1, outstandingrect-1, rectangle-500, and outstandingrect-500 expand on a randomly-generated 11-puzzle instance. The "y-axis" is the depth and the "x-axis" is the number of nodes expanded. The full visualization is shown and a portion outlined in red is zoomed in to show detail. The color of a cell represents when that node was expanded during the search — lightest pink was expanded first and darkest green was expanded last. The black dots show where goals were found.

It can be seen that outstanding-2 searches deeper than rectangle-1 but not as deeply as rectangle-500. Rectangle has a clean gradient because it follows its rigid expansion strategy, but outstanding jumps around between depth levels throughout the search, as expected. Outstandingrect searches very similarly to rectangle, except it focuses on shallower depths as the search progresses, probably due to the tie-breaking.

Additional Experiments

I tested changing two things from the previous experiment: the formula for discrepancy scores and the tie-breaking strategy of outstanding search across depths.

Relative Discrepancy Scores

Recall the definition of a discrepancy score:

$$discrepancyScore_n = d_n - d_{best}$$

I also tested *relative* discrepancy scores:

$$discrepancyScore_n = (d_n - d_{best})/d_{best}$$

Figure 16 shows the anytime performance of outstanding-2 with relative discrepancy scores in tiles compared to rectangle-1 and rectangle-500. Whereas with absolute discrepancy scores, outstanding-2 does well in this domain,

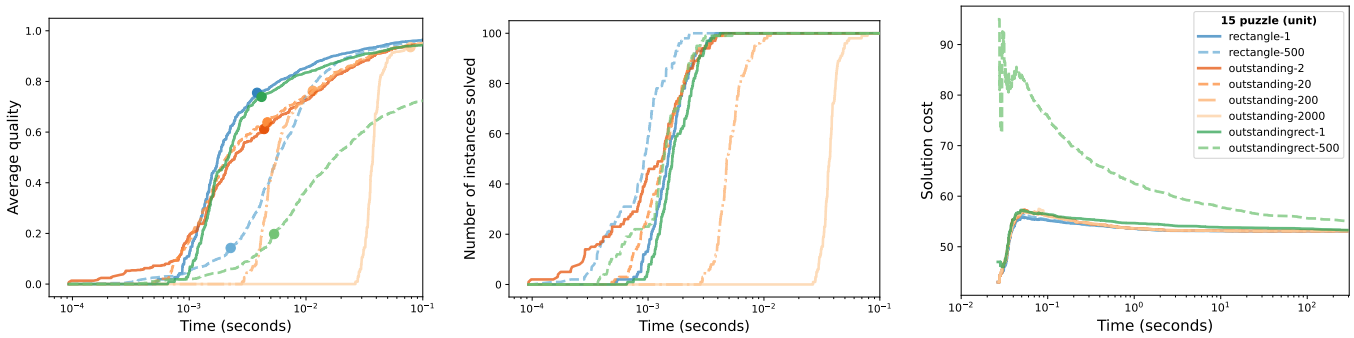


Figure 2: Anytime performance in **tiles (unit)** domain.

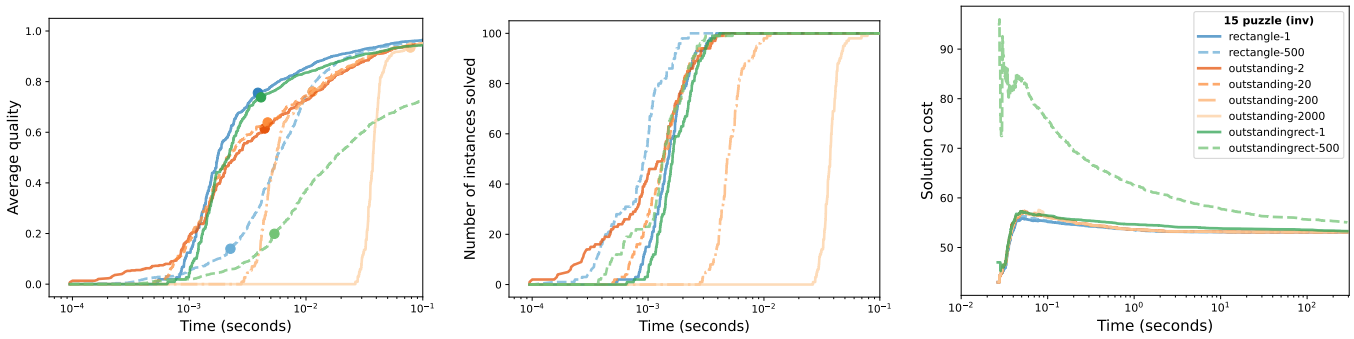


Figure 3: Anytime performance in **tiles (inv)** domain.

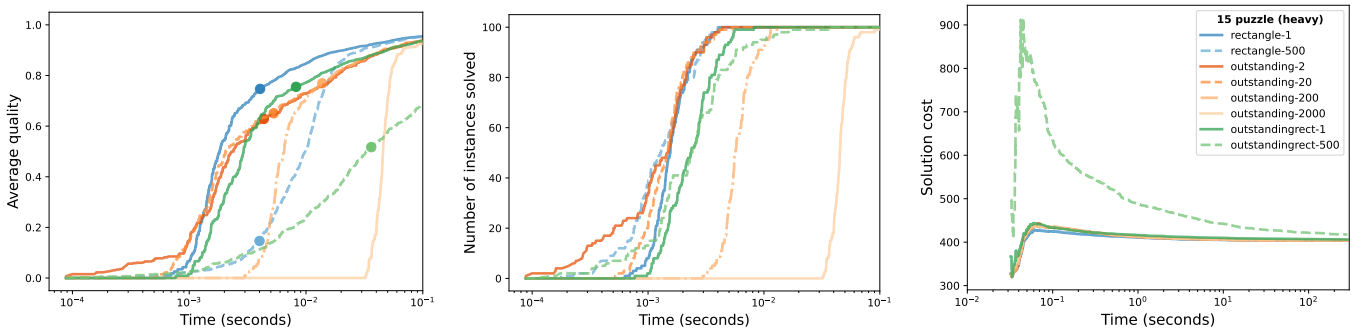


Figure 4: Anytime performance in **tiles (heavy)** domain.

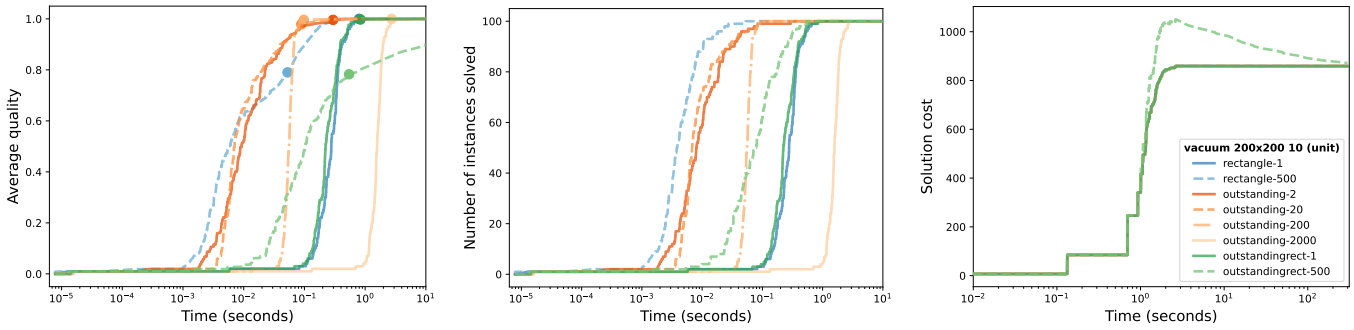


Figure 5: Anytime performance in **vacuum (unit)** domain.

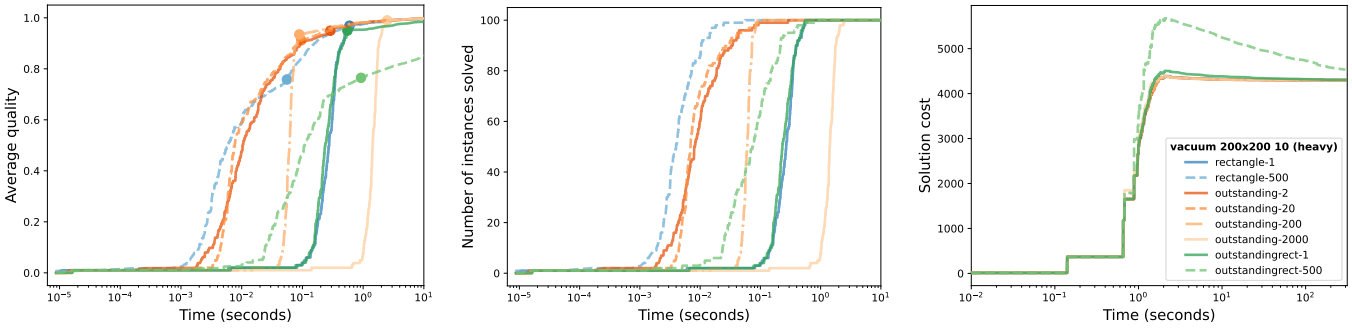


Figure 6: Anytime performance in **vacuum (heavy)** domain.

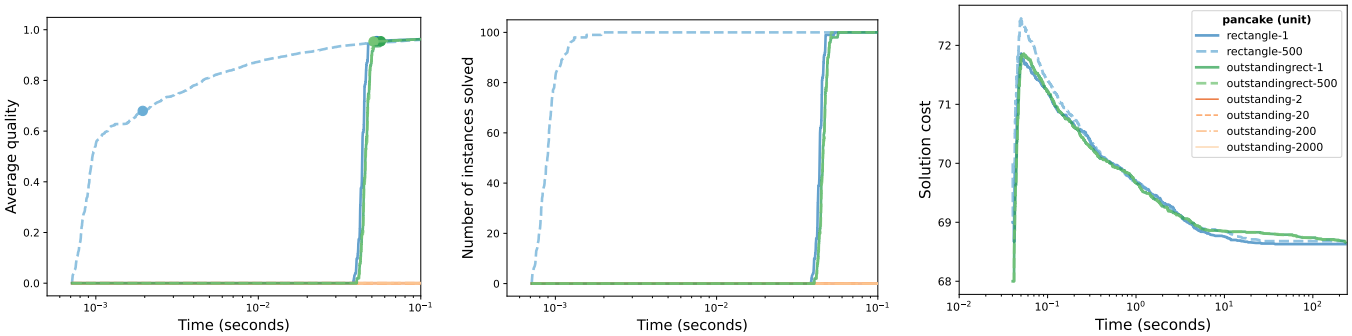


Figure 7: Anytime performance in **pancake (unit)** domain.

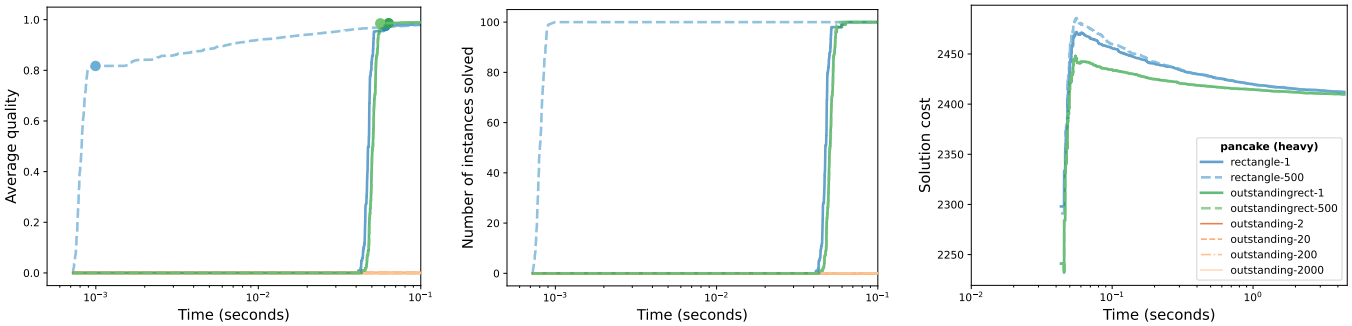


Figure 8: Anytime performance in **pancake (heavy)** domain.

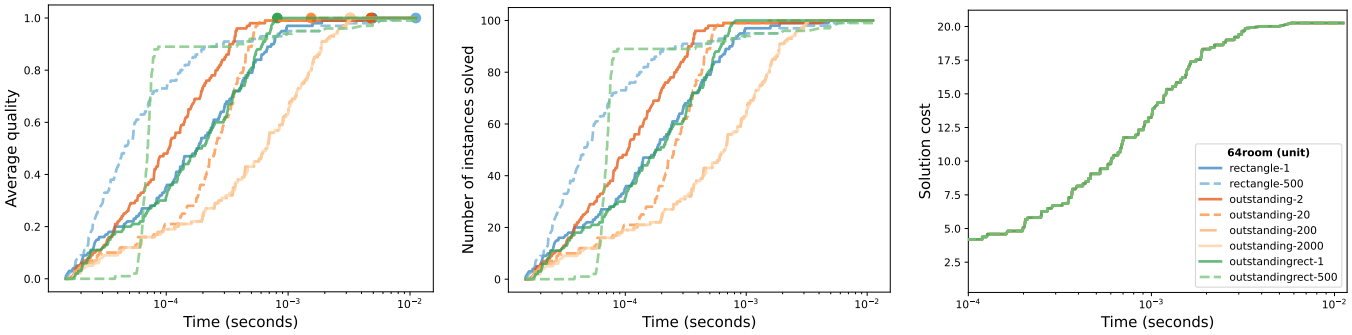


Figure 9: Anytime performance in **64room (unit)** domain.

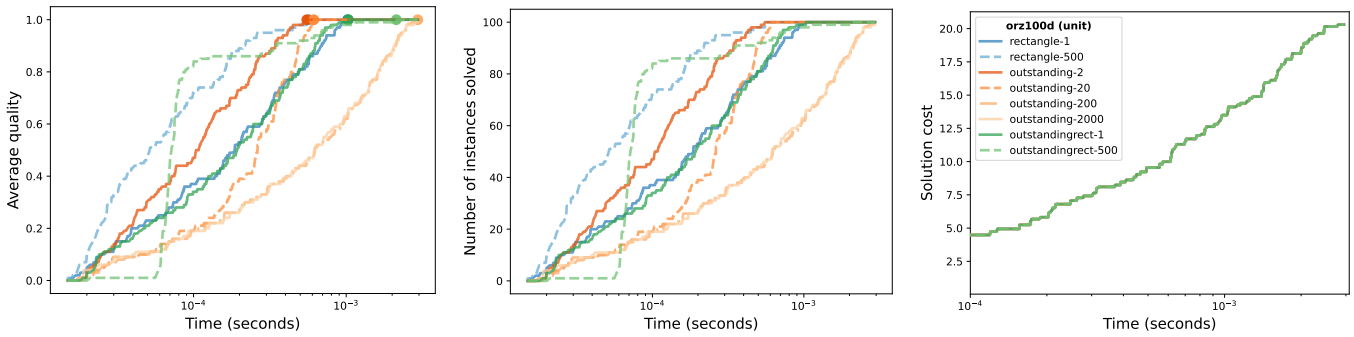


Figure 10: Anytime performance in **orz100d (unit)** domain.

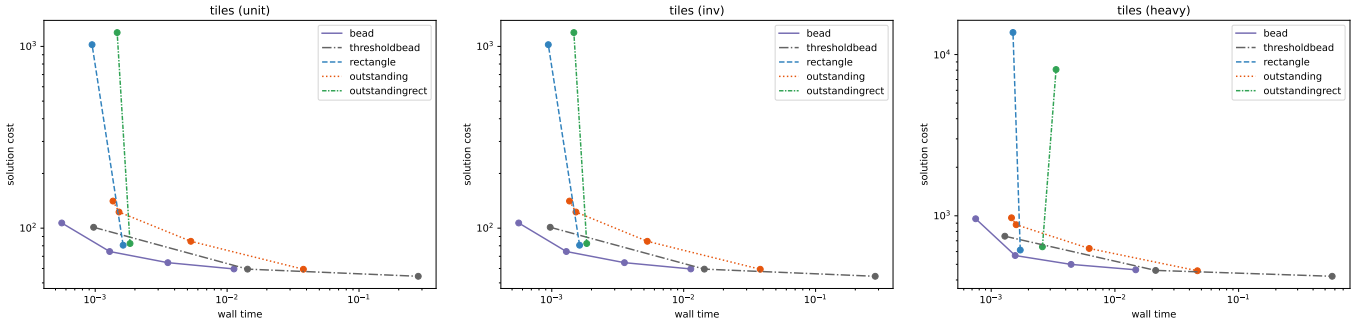


Figure 11: First solution performance in **tiles** domain.

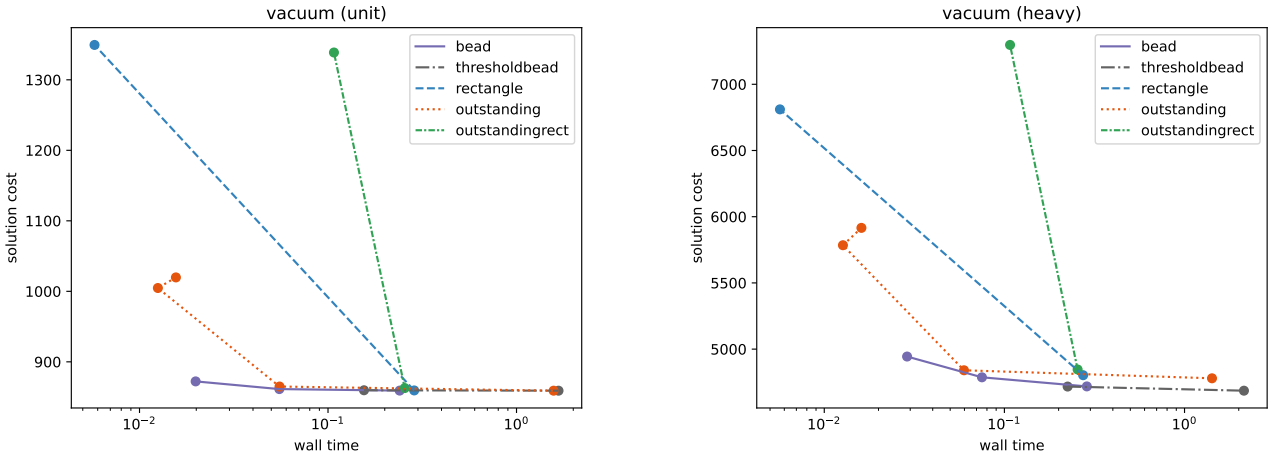


Figure 12: First solution performance in **vacuum** domain.

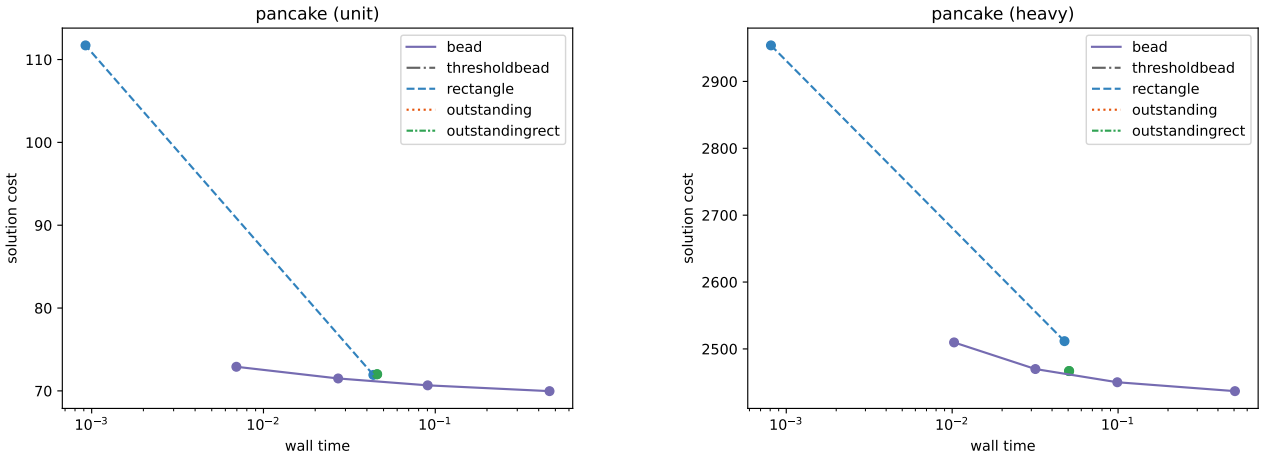


Figure 13: First solution performance in **pancake** domain.

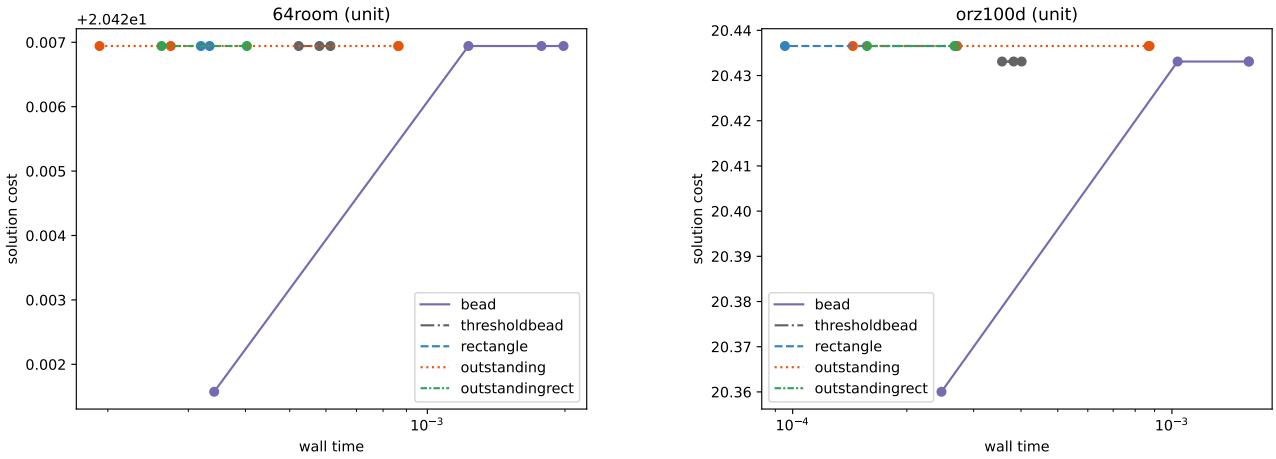
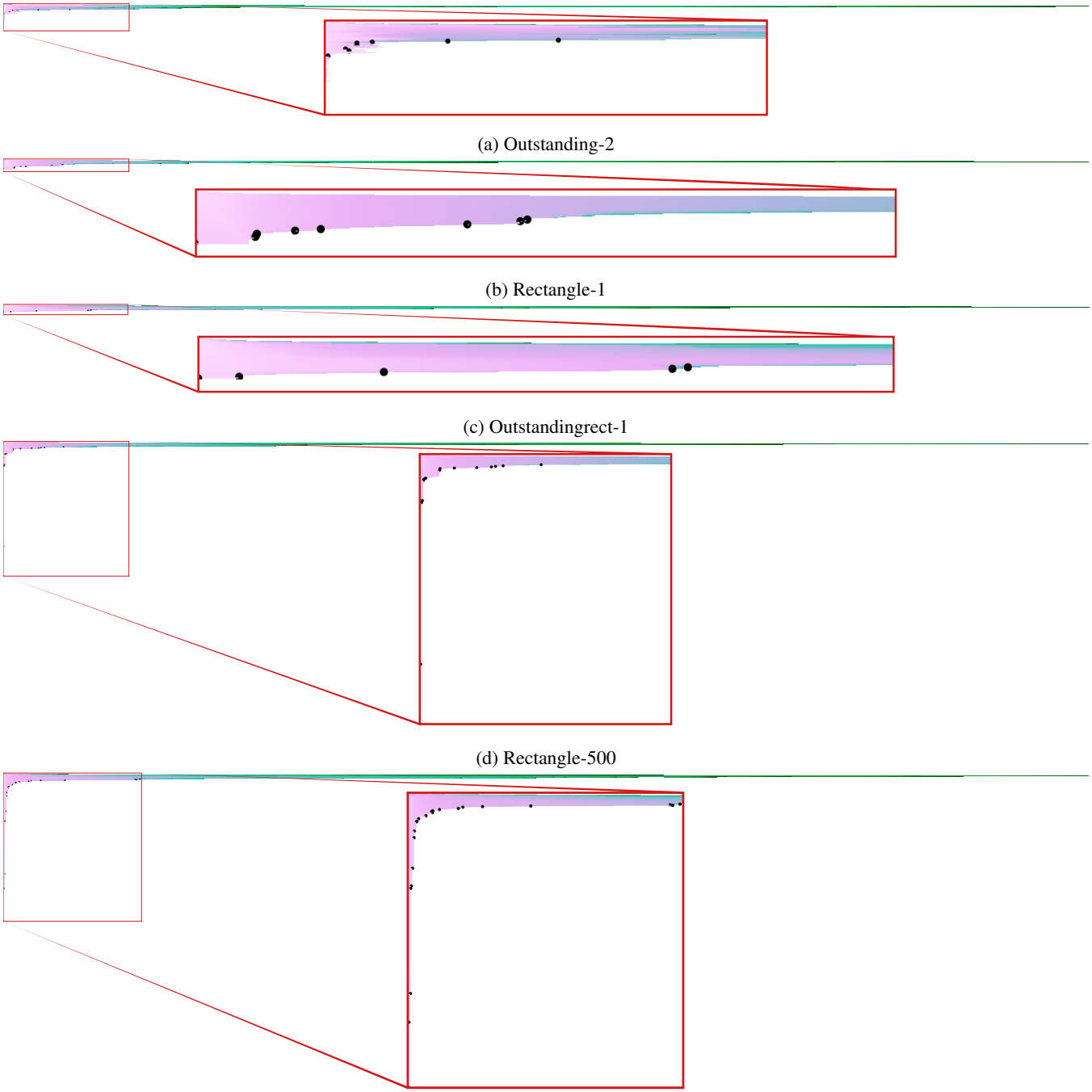


Figure 14: First solution performance in grid pathfinding domains **64room** and **orz100d**.



(a) Outstanding-2

(b) Rectangle-1

(c) Outstandingrect-1

(d) Rectangle-500

(e) Outstandingrect-500

Figure 15: Visualizations of how outstanding-2, rectangle-1, outstandingrect-1, rectangle-500, and outstandingrect-500 search on the same 11-puzzle instance.

it does terribly with relative, not even able to solve all instances in time. It performs worse in vacuum as well, and no better in the grid pathfinding domains.

Threshold bead was also unable to solve instances when using relative discrepancy scores.

Alternate Tie-Breaking

Recall that outstanding search (and outstanding rectangle search) breaks ties across depth levels by the shallower depth. The idea was to expand more nodes at shallow depths so that deeper depths would gradually get more (and hopefully better) options. However, I visualized how outstanding-2 was searching on a pancake instance, as can be seen in Figure 17a, and I noticed that it was searching in layers, always sticking to the shallowest depth when possible. This is why it would run out of memory before finding any solutions — it was going much too wide and not deep enough. I tried tie-breaking by deeper depth instead, and the behavior completely changed, as can be seen in Figure 17b. This time, outstanding rapidly went deep enough to find solutions.

Figure 18 shows that outstanding-2 with this deep tie-breaking scheme performs quite competitively in the pancake domain, whereas before it could not solve any instances. Outstandingrect-1 also outperforms rectangle-1, which it did not before. The same phenomena are true in the grid domains, as can be seen in Figure 19 and Figure 20, except outstandingrect-500 does much worse than before (and outstanding does not find all solutions in 64room). However, outstanding's outstanding performance does not extend to the tiles and vacuum domains, where it does worse than with shallow tie-breaking, as can be seen in Figure 21 and Figure 22. Outstandingrect also performs worse in these domains.

Discussion

Threshold bead fails to beat bead and outstanding rectangle fails to beat rectangle in most tested domains, so they do not appear to be promising alternatives. On the other hand, outstanding search does stand out — not as an algorithm that is superior to rectangle across domains, but as one that is more consistent with a single parameter value. While rectangle-1 is superior to rectangle-500 in some domains (tiles) and the opposite is true in others (vacuum, pancakes, and grid pathfinding), with the reasonable cautiousness value of $k = 2$, outstanding performs better than the weaker of the two rectangle searches across domains, and, in vacuum, even performs similarly to the stronger of the two. Outstanding also finds decent first solutions in comparison to rectangle.

Of course, outstanding also utterly fails on the pancake domain. Unless, that is, deep tie-breaking is used, in which case it has superior performance in the pancake and grid domains but worse performance in tiles and vacuum. This is a promising avenue for future research — automatic methods of setting the tie-breaking strategy based on instance features could have potential, or even just alternating between shallow and deep tie-breaking.

For threshold bead, it would be interesting to try having an *average beam width* parameter and let the algorithm determine the threshold it should set to approximate that average beam width for a given instance. It would also be useful to find out if small discrepancies in d are even informative about which node is better in different domains.

Conclusion

In this paper, I introduced the novel beam search variants threshold bead search, outstanding search, and outstanding rectangle search, the latter two of which are anytime algorithms based on rectangle search. These algorithms are more flexible than bead and rectangle search, but, from this initial exploration, that flexibility does not lead to better performance. However, outstanding search does have more consistent performance with an easy-to-set cautiousness parameter than rectangle search with its difficult-to-set aspect ratio parameter, as well as superior performance on some domains when using an alternate tie-breaking strategy, so it has potential.

References

Harvey, W. D.; and Ginsberg, M. L. 1995. Limited Discrepancy Search. In *Proceedings of IJCAI-95*.

Lemons, S.; López, C. L.; Holte, R. C.; and Ruml, W. 2022. Beam Search: Faster and Monotonic. In *Proceedings of AAAI-22*.

Lemons, S.; Ruml, W.; Holte, R.; and Linares Lopez, C. 2024. Rectangle Search: An Anytime Beam Search. In *Proceedings of AAAI-24*.

Likhachev, M.; Gordon, G. J.; and Thrun, S. 2003. ARA*: Anytime A* with Provable Bounds on Sub-Optimality. In *Proceedings of NIPS-03*.

Newell, A. 1978. Harpy, Production Systems and Human Cognition. In *Proceedings of Symposium on Cognition '78*.

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

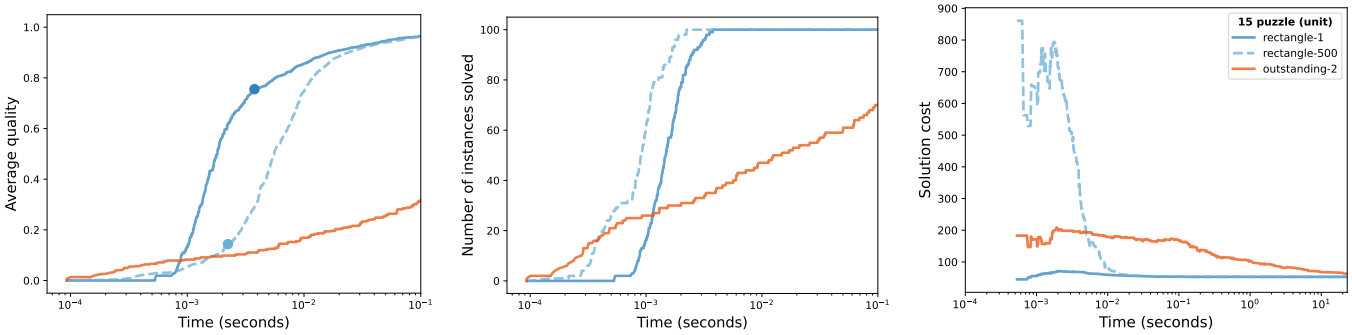
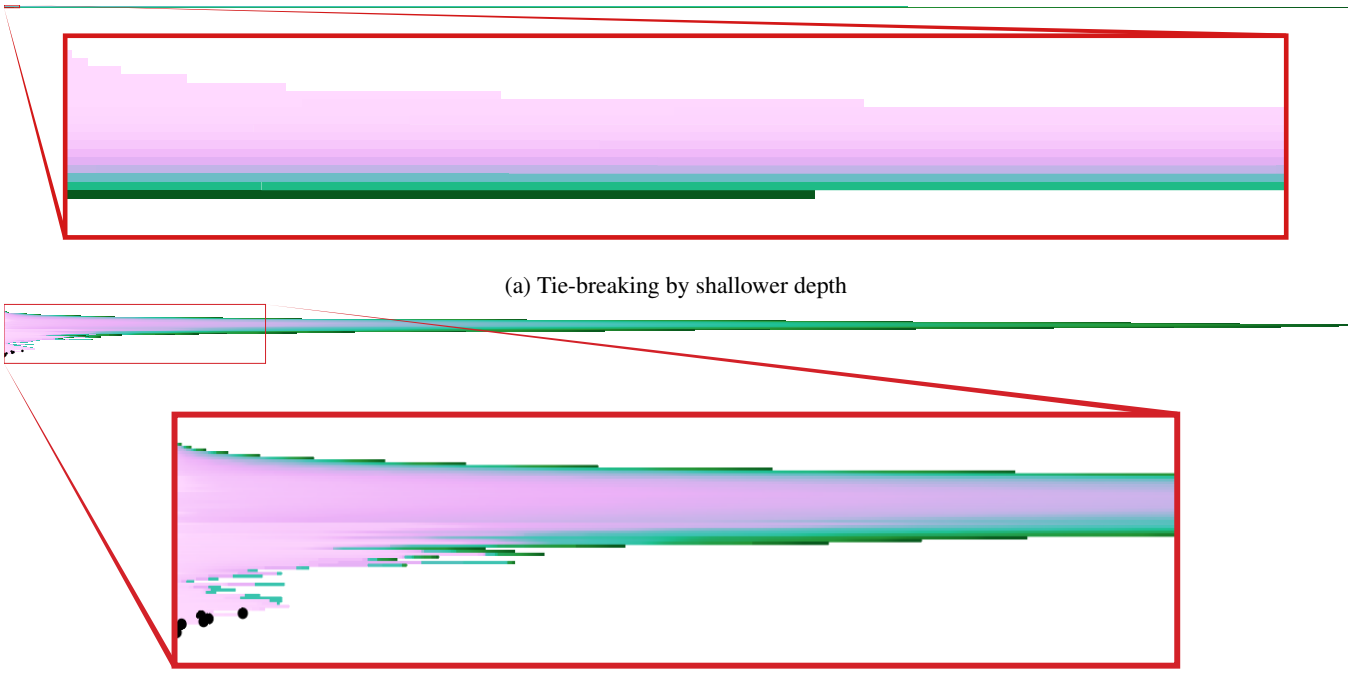


Figure 16: Anytime performance in **tiles (unit)** domain with relative discrepancy scores.



(a) Tie-breaking by shallower depth

(b) Tie-breaking by deeper depth

Figure 17: Visualizations of how outstanding-2 searches on a pancake instance with different kinds of tie-breaking across depths.

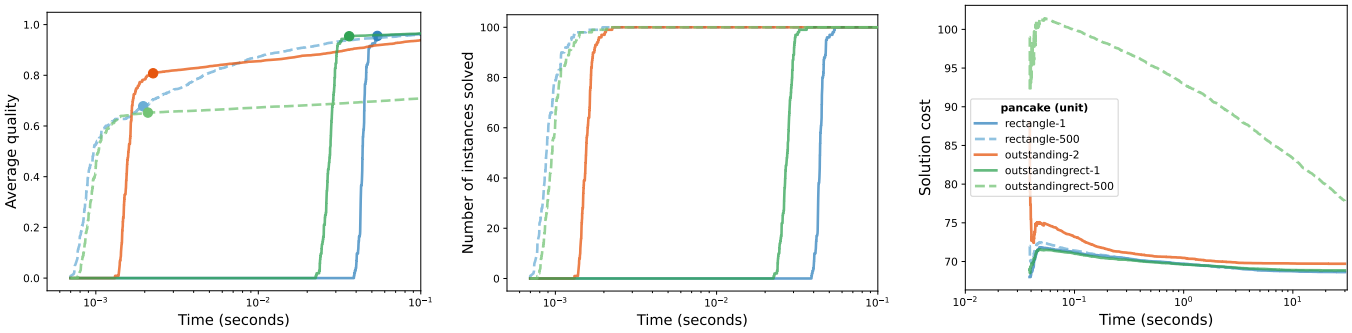


Figure 18: Anytime performance in **pancake (unit)** domain with tie-breaking by deeper depth.

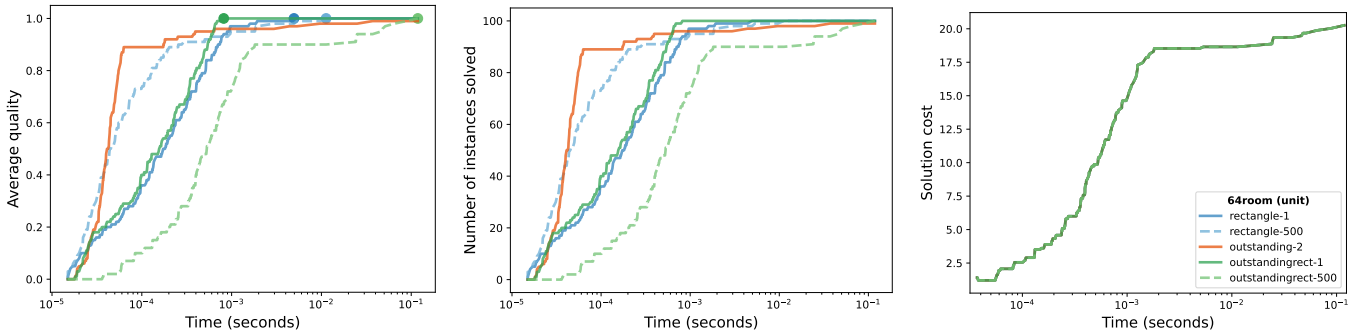


Figure 19: Anytime performance in **64room (unit)** domain with tie-breaking by deeper depth.

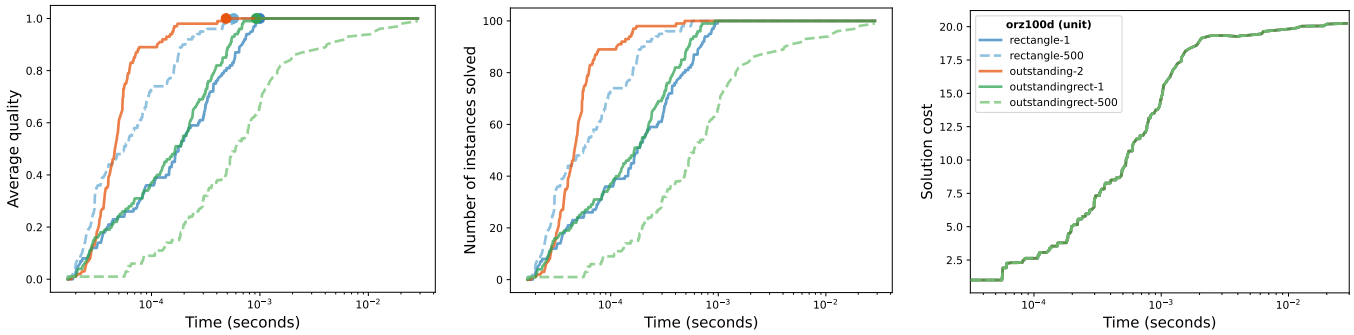


Figure 20: Anytime performance in **orz100d (unit)** domain with tie-breaking by deeper depth.

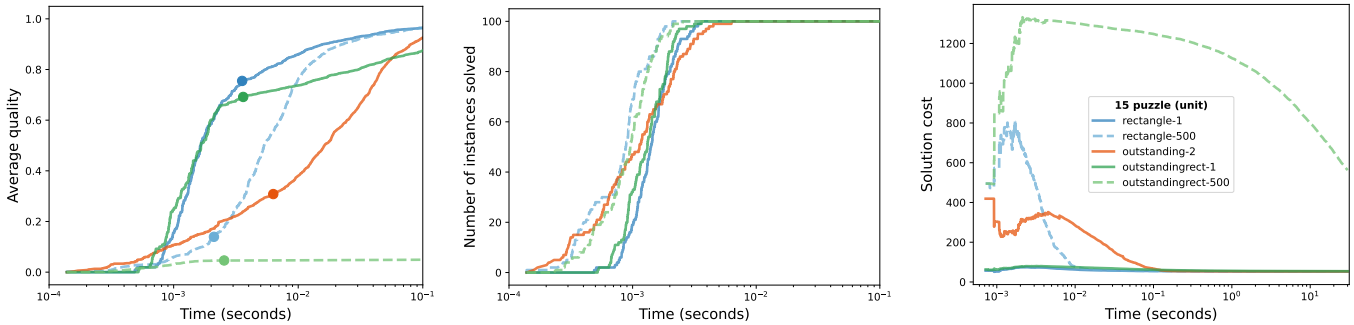


Figure 21: Anytime performance in **tiles (unit)** domain with tie-breaking by deeper depth.

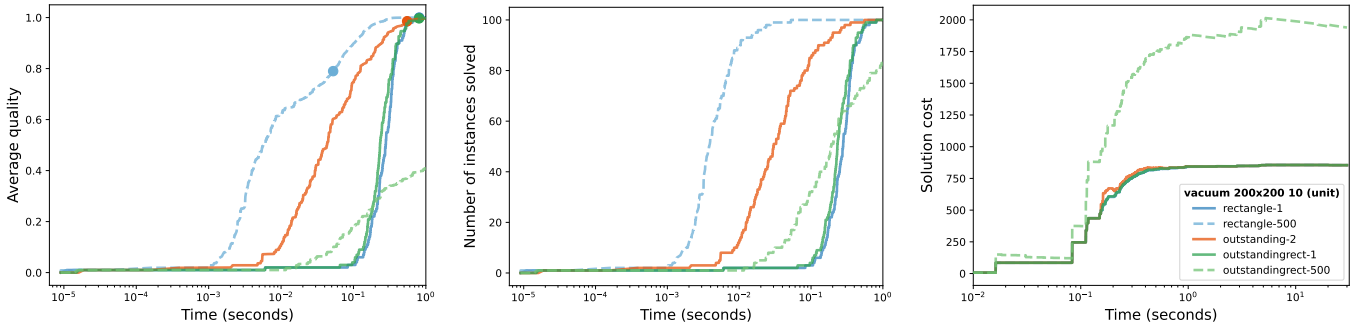


Figure 22: Anytime performance in **vacuum (unit)** domain with tie-breaking by deeper depth.