Dynamic Potential Search – A New Bounded Suboptimal Search Algorithm

Daniel Gilon
ISE Department
Ben-Gurion University
Be'er-Sheva, Israel
(gilond@post.bgu.ac.il)

Ariel Felner
ISE Department
Ben-Gurion University
Be'er-Sheva, Israel
(felner@bgu.ac.il)

Roni Stern
ISE Department
Ben-Gurion University
Be'er-Sheva, Israel
(roni.stern@gmail.com)

Abstract

Potential Search (PS) is an algorithm that is designed to solve bounded cost search problems. In this paper, we modify PS to work within the framework of bounded suboptimal search and introduce Dynamic Potential Search (DPS). DPS uses the idea of PS but modifies the bound to be the product of the minimal f-value in OPEN and the required suboptimal bound. We study DPS and its attributes. We then experimentally compare DPS to WA* and to EES on a variety of domains and study parameters that affect the behavior of these algorithms. In general we show that in domains with unit edge costs (e.g., many standard benchmarks) DPS significantly outperforms WA* and EES but there are exceptions.

1 Introduction

Best-first search algorithms maintain an OPEN list of nodes throughout the search. At each *expansion cycle* the "best" node from OPEN is chosen and expanded, i.e., removed from OPEN and its children are generated and are inserted to OPEN. Best-first search algorithms differ in their *evaluation function*, which chooses which node in OPEN is considered the "best". The A* algorithm (Hart, Nilsson, and Raphael 1968) is a well-known *best-first search* algorithm. Its evaluation function is $f(n) = g(n) + h(n)$, where $h(n)$ is a heuristic function estimating the cost from n to a goal node. If $h(n)$ is *admissible* (i.e., is always a lower bound) then A* is guaranteed to find an optimal (lowest-cost) solution.

However, some problems require a large amount of computing resources (i.e., both time and memory). In addition, some applications (e.g., video games, embedded systems or mobile apps) significantly restrict the amount of time (sometimes below 1ms (Bulitko et al. 2011)) or restrict the memory that is allowed for the problem solving. In such cases, one must settle for a suboptimal solution by trading time/memory for solution quality. Moreover, sometimes an optimal solution is not necessary and the aim is to minimize some other objective. Two such non-optimal search settings are the following:

(1) Bounded suboptimal search (denoted here as BSS). In BSS(B) we are given a bound B and the task is to find a solution with cost $\leq B \times P_{opt}$ where P_{opt} is the cost of the optimal solution.

(2) Bounded cost search (Stern et al. 2014) (denoted here as BCS). In BCS(C) we are given a cost C and the task is to find a solution with cost $\leq C$.

Potential Search (PS) (Stern et al. 2014) is an algorithm specifically designed for BCS. PS is a best-first search algorithm that chooses to expand the node n from OPEN with the largest $u(n) = \frac{C-g(n)}{h(n)}$. $u(n)$ is also known as the "potential" of n and under certain conditions, the node with the largest $u(n)$ is the node that is most likely to be part of a solution with cost $\leq C$.

In this paper we show a general way to migrate algorithms from BCS to BSS and vice versa, given some conditions. Based on that we modify PS to the BSS setting and introduce a new algorithm called *dynamic potential search* (DPS). In PS the cost bound C remains constant throughout the search. In DPS we dynamically modify C and always set $C = f_{min} \times B$. Then, we expand the node in OPEN with the largest potential with respect to the current C. We prove that DPS is a special case of *focal search* and is thus guaranteed to find a solution within the desired suboptimality bound.

A weakness of DPS is that the priority of a node in OPEN depends on f_{min}, which increases throughout the search. Thus, OPEN is reordered every time this happens. We introduce a bucket-based data structure that remedies this. Each node is associated with a bucket based on its (g, h) pair. Then, we only need to reorder these buckets.

Finally, we experimentally compare DPS to other known algorithms that are designed for BSS, namely to WA* (Pohl 1970) and to EES (Thayer and Ruml 2011). We study some parameters that influence the behavior of these algorithms. In general, in domains with non-uniform edge costs, it is beneficial to use EES as it commonly outperforms both WA* and DPS (but not always). However, in domains with unit edge costs such as many of the traditional testbeds in the field of heuristic search (e.g., combinatorial puzzles, grid based path finding etc.) DPS significantly outperforms EES and WA* by a factor of up to 180 in the number of nodes expanded and in the CPU time. Exceptions exist here, too.

2 Background and Previous Work

Suboptimal search algorithms can be divided into a number of classes based on the different guarantees they provide on the quality of the solution they return. Next, we discuss these

classes and related existing algorithms.

2.1 Any Solution

Perhaps the simplest class of suboptimal algorithms has no constraint on the solution quality: any solution is acceptable. Greedy-best-first search (GBFS, AKA *pure heuristic search)* is perhaps the most famous algorithm for this purpose. At each expansion cycle, the node with the minimal h in OPEN is expanded. A more recent example is *Speedy search* (Wilt and Ruml 2014), which is similar to GBFS but uses d instead of h, where d is the estimated number of edges (without considering the weights of the edges) to the closest goal. The rationale behind Speedy search is that there is no constraint on the cost of the returned solution, so it can focus on finding a solution with minimal search effort regardless of its cost. Thus, Speedy search only considers the number of edges until the goal is found and ignores their weights. Of course, in unit-cost domains (where all edges weigh 1) d and h are identical, and thus Speedy and GBFS are equivalent.

Sometimes finding any solution is not enough as there are restrictions on the quality of the solution that is acceptable. We next describe two important classes that are the main focus of this paper.

2.2 Bounded Suboptimal Search

In *bounded suboptimal search* (denoted by BSS) we have a constraint on the relation between the cost of the returned solution and the optimal solution cost (P_{opt}). Formally: let $B \geq 1$ be a given constant bound. In BSS(B) the task is to find a solution with cost $\leq B \times P_{opt}$. We say that a solution is *B-admissible* if it satisfies this requirement[1] and an algorithm is called a *BSS algorithm* if it is guaranteed to return a B-admissible solution.

2.2.1 Focal Search A large number of BSS algorithms have been proposed over the years (see a nice survey by Thayer and Ruml, 2011). Virtually all of these algorithms can be viewed as implementations of a general search framework that is sometimes called Focal Search (Pearl and Kim 1982; Ebendt and Drechsler 2009; Valenzano et al. 2013).[2]

Focal Search is a special case of best-first search that maintains two sets of nodes. Similar to any best-first search, it maintains OPEN which includes all nodes that were generated but not expanded. In addition, Focal Search maintains a list of nodes FOCAL $\subseteq$ OPEN.

Definition 1 (Focal) *Let f_{min} be the minimal f-value in* OPEN. FOCAL $= \{n \in \text{OPEN} | f(n) \leq B \times f_{min}\}$

As a special case of best-first search, a Focal Search algorithm chooses in every step a single node from OPEN and expands it, but it is constrained to only choose to expand a node that is also in FOCAL. As the search progresses, f_{min} may increase. When this occurs, the range $B \times f_{min}$ also grows and more nodes from OPEN are added to FOCAL. This allows a broader range of nodes to be considered for expansion in subsequent expansion cycles. Once a goal node is chosen for expansion, the search halts. When h is consistent and duplicate detection is performed any Focal Search is a BSS algorithm. This is because f_{min} is a lower bound on P_{opt} and when a goal node t is expanded it must be in FOCAL, and thus $f(t) = g(t) \leq B \times f_{min} \leq B \times P_{opt}$. Algorithm 1 presents the main structure of Focal Search. Different Focal Search algorithms differ in which node from FOCAL they choose to expand (line 5).

Algorithm 1: Focal Search: main procedure

```
1  focal-search(start state S)
2      OPEN ← {S};
3      FOCAL ← {S};
4      while FOCAL ≠ ∅ do
5          best ← ChooseNode(FOCAL)
6          Remove best from FOCAL and OPEN
7          if best is a goal then return best;
8          if f_min increased then FixFocal() ;
9          for n ∈ neighbors(best) do
10             Add n to OPEN
11             if f(n) ≤ B × f_min then add n to FOCAL ;
12         end
13     end
14 end
```

Perhaps the simplest form of Focal Search is to choose the node in FOCAL with the smallest h-value. This algorithm is usually referred to as A^*_ϵ (Ebendt and Drechsler 2009) although the original A^*_ϵ (Pearl and Kim 1982) is actually more general, allowing other rules to choose from FOCAL.

In its basic form, Focal Search must keep FOCAL as a separate list. In this case, any time f_{min} increases, the relevant nodes from OPEN are identified and are added to FOCAL. This is done by `FixFocal` (line 8 in Algorithm 1) which incurs some computation overhead. However, in some cases it is possible to implement Focal Search without maintaining a separate list of nodes for FOCAL. In these cases the decision rule for which node to expand from OPEN guarantees that the node n chosen for expansion is in FOCAL, i.e., that $f(n) \leq B \times f_{min}$. A prominent example of a BSS algorithm that does not explicitly maintain FOCAL is Weighted A* (Pohl 1970; 1973) discussed next.

2.2.2 Weighted A* (WA*) is perhaps the most famous and simple BSS algorithm. WA* is a best-first search algorithm that prioritizes nodes in OPEN according to $f_W(n) = g(n) + W \cdot h(n)$, where $g(n)$ is the cost of the lowest known path from the start state to n, $h(n)$ is an admissible heuristic of reaching the goal from n and $W \geq 1$. When $W = 1$, WA* is identical to A*. When $W \to \infty$, WA* converges to GBFS. It was proven that the solution returned by WA* when setting $W = B$ is B-admissible (Pohl 1973).

WA* is a special case of Focal Search as was noted by (Ebendt and Drechsler 2009). Let $best$ be the node cho-

[1] Sometimes B is written as $B = 1 + \epsilon$ (Ebendt and Drechsler 2009; Pearl and Kim 1982) and the related term is ϵ-admissible. Similarly, some authors use the term W instead of B. We choose to use B to differentiate between the desired bound B and the weight W used by the search algorithms like WA*.

[2] Focal Search was first called A^*_ϵ (Pearl and Kim 1982). However, nowadays A^*_ϵ is associated with a special case of Focal Search in which nodes in FOCAL are chosen according to their h value.

sen for expansion (i.e., $f_W(best)$ is minimal) and let x be a node whose $f(x) = f_{min}$. It is easy to see that $f(best) \leq f_W(best) = g(best) + W \cdot h(best) \leq g(x) + W \cdot h(x) \leq W \cdot f(x) = W \cdot f_{min}$. Therefore, using f_W does not require to actually maintain FOCAL in a separate list as all nodes expanded by WA* must be in FOCAL.

2.2.3 Explicit Estimation Search *Explicit estimation search* (Thayer and Ruml 2011) is a recently introduced BSS algorithm. EES uses two heuristics: h (estimation of the cost to the goal) and d (estimation of the number of edges to the goal). However, since h and d must be admissible (underestimating) they might be inaccurate. Thus, EES uses two other inadmissible estimates $\hat{h}$ and $\hat{d}$ which are supposed to be more accurate than d and h since they are not restricted to be admissible. While one may come up with any functions for $\hat{h}$ and $\hat{d}$, Thayer and Ruml (2011) use $\hat{h}$ and $\hat{d}$ that are learned from known mistakes of h and d during the search. We use their method in the experiments below. EES has specific rules based on $\hat{h}$ and $\hat{d}$ as to which node from FOCAL to expand next based on the following definitions:

$$best_f = \underset{n \in OPEN}{\operatorname{argmin}} f(n) \quad (i.e., f(best_f) = f_{min}) \tag{1}$$

$$best_{\hat{f}} = \underset{n \in OPEN}{\operatorname{argmin}} \hat{f}(n) \tag{2}$$

$$best_{\hat{d}} = \underset{n \in OPEN \wedge \hat{f}(n) \leq w \cdot \hat{f}(best_{\hat{f}})}{\operatorname{argmin}} \hat{d}(n) \tag{3}$$

where $\hat{f}(n) = g(n) + \hat{h}(n)$ and $w = B$.

At every expansion, EES chooses from among these three nodes using the following rule:

1. **if** $\hat{f}(best_{\hat{d}}) \leq w \cdot f(best_f)$ **then choose** $best_{\hat{d}}$
2. **else if** $\hat{f}(best_{\hat{f}}) \leq w \cdot f(best_f)$ **then choose** $best_{\hat{f}}$
3. **else choose** $best_f$

We note that $\hat{f}(n) \geq f(n)$ and thus the conditions in items 1 and 2 guarantee that at all times the node chosen from expansion n has $f(n) \leq \hat{f}(n) \leq w \cdot f(best_f) = B \times f_{min}$. Thus EES is also a BSS algorithm that is based on Focal Search.

EES was shown to perform very well on many domains for the BSS setting (Thayer and Ruml 2011). However, EES has a few weaknesses. First, in order to identify the three nodes defined above ($best_f$, $best_{\hat{f}}$ and $best_{\hat{d}}$) EES sorts all generated nodes in three different OPEN lists, based on f, $\hat{d}$ and $\hat{h}$.[3] Therefore, the constant or logarithmic time per node of maintaining these lists is three times more than a search with only one OPEN list such as WA*. Second, having strong $\hat{h}$ and $\hat{d}$ functions is challenging. Thayer and Ruml (2011) provide a number of methods to obtain these functions by learning them during the progress of the search. But these methods are not trivial. Third, due to the three OPEN lists and the non-trivial heuristic functions used, EES is rather complex to implement.

[3]EES does not maintain an explicit FOCAL as it chooses values based on $\hat{h}$ or $\hat{d}$ but it only *checks* whether they are in FOCAL. This can be done by just knowing B and f_{min}.

2.3 Bounded Cost Search

Bounded cost search (BCS) (Stern et al. 2014) is another setting for search problems. In BCS we have no knowledge of the cost of the optimal solution and are indifferent to it. Instead, in BCS(C) we are given a cost C and the task is to find a solution with cost $\leq C$. Stern et al. (2014) give a number of scenarios for BCS. For example, one might have a limited budget and would like to find a solution within the budget as fast as possible.

2.3.1 Potential Search (PS) (Stern, Puzis, and Felner 2011; Stern et al. 2014) is an algorithm specifically designed for BCS. PS is a best-first search algorithm which chooses to expand the node n from OPEN with the largest "potential", $u(n)$, which is defined as $u(n) = \frac{C-g(n)}{h(n)}$.[4] In addition, for an admissible h, the algorithm prunes any node n for which $f(n) = g(n) + h(n) > C$, as it will not lead to a solution within the bound.

Intuitively, given a node n, $C - g(n)$ is an upper bound on any path in the search tree below node n that will still be within the bound C. Dividing this by the estimated cost to the goal, $h(n)$, gives its *potential*, i.e., how likely there is a goal node within the bound in the subtree rooted at n. Stern et al. (2014) showed that under certain conditions PS expands in every iteration the node that is most likely to be part of a solution that is within the bound. PS will find a desired solution if one exists. PS was shown to be very effective in finding BCS solutions.

2.3.2 Bounded Cost EES Thayer et al. (2012) developed two variants of EES for BCS called BEES and BEEPS. These algorithms have different ways of using the rules of EES to choose which node from OPEN to expand next while assuring that the solution returned is within the bound C. BEES and BEEPS were shown to perform better than PS, especially on non-unit edge cost domains.

3 Migration Between BSS and BCS

We say that a BCS algorithm is *reasonable* if it has the following two attributes: **(1)** it has a best-first structure (i.e., OPEN and an expansion rule). **(2)** any node n that is generated with $f(n) > C$ is pruned and not added to OPEN as it may never lead to a solution $\leq C$.

We now show that BSS and BCS have relatively similar structure and thus any Focal Search algorithm (for BSS) may be modified to work for BCS. Similarly, any reasonable BCS algorithm may be modified to work for BSS as a Focal Search.

We note that in both settings we aim to find a solution with cost *smaller than or equal to* some quantity ($B \times P_{opt}$ for BSS and C for BCS). FOCAL is used in BSS(B) to ensure that any node chosen from it has f-value below the bound. Thus, a Focal Search algorithm may choose any node from FOCAL. Similarly, we may define a similar list for reasonable BCS(C) algorithms that includes all nodes n in OPEN with $f(n) \leq C$. We call this list of nodes FOCAL(C).

[4]For cases where $h(n) = 0$, $u(n)$ is defined to be ∞, causing such nodes to be expanded first.

Algorithm 2: Focal Search for BCS: main procedure

```
1  focal-search(start state S)
2      FOCAL ← {S};
3      while FOCAL ≠ ∅ do
4          best ← ChooseNode(FOCAL)
5          Remove best from FOCAL
6          if best is a goal then return best;
7          for n ∈ neighbors(best) do
8              if f(n) ≤ C then add n to FOCAL;
9          end
10     end
11 end
```

Given this definition of FOCAL(C), one can migrate any reasonable BCS to a Focal Search BSS algorithm and vice versa. The general framework of maintaining FOCAL is a little different. In FOCAL for BSS, the upper threshold for nodes from OPEN ($B \times f_{min}$) is dynamically changing and depends on the changes of f_{min}. By contrast the upper threshold (C) for FOCAL(C) remains constant. Nevertheless, the same decision rule (`ChooseNode`, line 5 of Algorithm 1) of which node to expand from the current FOCAL can migrate between these frameworks.

A general reasonable BCS algorithm which is built this way is shown in Algorithm 2. It has the same structure as Algorithm 1 but the maintenance of FOCAL is different as just defined. The core of these algorithms is the `ChooseNode`(FOCAL) (line 4 of Algorithm 2) which can migrate between the two frameworks.

The different algorithms are summarized in Table 1. In fact, Thayer et al. (2012) transformed the concept of EES, originally developed for BSS, to BCS exactly along these lines (the up arrow in the table). In this paper, based on our general understanding we are completing the picture and migrating PS from BCS to BSS.[5]

4 Dynamic Potential Search

We now introduce our new algorithm, *Dynamic Potential Search* (DPS). Based on our observation on the transformation of `ChooseNode`(), we transform PS from BCS to BSS. PS can be implemented using Algorithm 2. DPS is implemented using Algorithm 1 but uses the same `ChooseNode`() rule as PS.

DPS uses a FOCAL list just as every Focal Search BSS algorithm. That is, DPS maintains: FOCAL $= \{n \in$ OPEN $\mid f(n) \leq B \times f_{min}\}$. Similar to PS we now choose the node n with the highest "potential", but now the potential relates to the likelihood of finding a node below n with cost $\leq B \times f_{min}$. In other words, given the current FOCAL, we use the same `ChooseNode`() as PS but set $C = B \times f_{min}$. Formally, DPS chooses to expand a node n from FOCAL that maximizes:

$$ud(n) = \frac{B \times f_{min} - g(n)}{h(n)}$$

[5]We note that while PS and EES can directly migrate from BCS to BSS and vice versa, WA* does not directly migrate to BCS because WA* does not directly define FOCAL in its pseudo code.

	FOCAL	EES	PS	WA*
BCS	$f \leq C$	BEES	**PS**	N/A
		$\Uparrow$	$\Downarrow$	
BSS	$f \leq B \times f_{min}$	EES	**DPS**	WA*

Table 1: The different algorithms

4.1 Theoretical Analysis

We now prove that similar to WA*, DPS does not need to maintain an explicit FOCAL and maintaining OPEN suffices. That is, when we choose a node with the maximal $ud(\dot{})$ from OPEN then this node is guaranteed to be inside FOCAL.

Let m be the node whose $f(m) = g(m) + h(m)$ is minimal in OPEN, i.e., $f_{min} = f(m)$. Let n be the node whose $ud(n) = \frac{(B \times f(m)) - g(n)}{h(n)}$ is maximal in OPEN. We want to prove that:

$$f(n) = g(n) + h(n) \leq B \times (g(m) + h(m))$$

We write:[6]

$$ud(m) = \frac{(B \times f(m)) - g(m)}{h(m)} = \frac{(B-1) \times g(m) + Bh(m)}{h(m)} = \frac{(B-1) \times g(m)}{h(m)} + B > B \geq 1$$

The leftmost fraction in the last line is nonnegative because all its internal terms are nonnegative.

Now, by definition $ud(n) \geq ud(m) \geq 1$.

So, $ud(n) = \frac{B \times f_{min} - g(n)}{h(n)} \geq 1$ this means that

$$B \times (g(m) + h(m)) \geq g(n) + h(n)$$

■

4.2 Challenge of Sorting OPEN

The priority function that is used to sort nodes inside an ordinary OPEN list remains fixed throughout the search. This is not the case for DPS because it sorts the entries according to $ud(n)$ which depends on f_{min} but f_{min} changes over time. So, every time f_{min} changes, we need to reorder OPEN to reflect the new priority of nodes. This may cause a significant overhead as every time this happens it may incur $O(N)$ operations where N is the number of nodes in OPEN. We note that all three algorithms discussed in this paper are special cases of Focal Search. However, DPS is unique in the sense that unlike EES and WA*, DPS has to reorder its OPEN every time f_{min} is increased.

We overcame this problem of DPS through the following mechanism (Burns et al. 2012). We note that each pair of values (g, h) is associated with the same ud-value. Therefore we implemented OPEN as a priority queue which maintains buckets of (g, h) pairs. Each bucket maintains a linked list of nodes. Therefore, every time f_{min} changes, we only need to reorder the relative position of the buckets while the list of nodes inside each bucket remains untouched. For unit-edge cost domains, let D be the diameter of the search space. The number of different (g, h) pairs is at most D^2. Let T be

[6]If $h(m) = 0$ then $ud(m) = \infty$. Thus, $ud(m) > 1$ as we want to prove now.

Domain	depth	h_0	Inc	Max-B	Max-nodes	Exp
15 puzzle	51	36	10	47	1,869,786	846,320
Dock robot	25	33	149	999	21,299	78,363
Vacuum	845	555	245	999	26,265	530,087
P40 GAP	39	36	0	0	0	885
P40 GAP-2	40	33	1	70	1,697,563	9,693

Table 2: The (g, h)-pair data structure effectiveness

the number of times during the search that f_{min} is changed. The overhead of keeping OPEN sorted at all times is therefore $O(T \times D^2)$. If the number of states in the domain is exponential in D then this is an exponential reduction.

Table 2 shows data from our experiments (details below) that impact the performance of the bucket-based OPEN. Each row corresponds to one of our benchmark domains and all the numbers are for $B = 1.1$. The *depth* column gives the depth in the search tree of the solution that was found. The h_0 column reports the h-value of the start state. The *Inc* column gives the number of times during the execution of DPS that f_{min} has increased. As can be seen, there is a strong correlation between these three measures. Vacuum is at the extreme case, with 245 increases of f_{min}. The next columns, *Max-B* and *Max-nodes* report the maximal numbers of buckets that existed at any given point where f_{min} increased and the maximal number of nodes at those times, respectively. In most cases, the ratio between these two measures shows a substantial reduction of up to 4 orders of magnitude. Relative to the number of expanded nodes, shown in the last column, the overhead of sorting the buckets is negligible.

5 Experimental Domains

We evaluated experimentally the performance of DPS on four search domains. We describe them next.

The 15 Puzzle The *15-puzzle* is a well-known benchmark. As a heuristic, we used the classical Manhattan Distance (MD). To provide more comprehensive experimental evaluation, we also experimented on two non-unit edge costs variants of this domain: the *heavy 15-puzzle*, in which moving tile $\#X$ costs X, and the *inverse 15-puzzle*, in which moving tile $\#X$ costs $1/X$. The MD heuristic can be easily modified for these variants (Thayer and Ruml 2011).

The Pancake Puzzle The K-pancake puzzle is another standard benchmark domain. The task is to sort a vector of numbers $V[K]$ where there are $K - 1$ operators, where operator i reverses a prefix of size $i + 1$. As a heuristic, we used the GAP heuristic (Helmert 2010) which adds 1 for every two adjacent numbers that are not consecutive (hence a gap). We created a *heavy* variant of this puzzle where the cost of the operator that flips a prefix $(V[1] \dots V[i + 1])$ is the maximum among the two elements on the extreme sides of the prefix, i.e., $\max(V[1], V[i + 1])$.[7] As an admissible heuristic, we generalized the GAP heuristic to what we refer to as HGAP. For each gap between elements x and y, in HGAP we add $\min(x, y)$ to the heuristic instead of just 1 as in ordinary GAP.

The Vacuum Cleaner Domain This domain was introduced by Thayer and Ruml (2011) and is inspired by the first state-space presented in Russell and Norvig's (1995) textbook. A vacuum cleaner is working in a grid (200×200) with obstacles (35% of the cells) and there are a number of dirt spots. The cleaner should find a a tour that cleans all dirty spots. When carrying dirt, the cleaner becomes heavier and therefore every dirty spot that was cleaned adds 1 to the cost of moving the cleaner. This domain resembles the traveling salesman problem (TSP) and we used a heuristic that is based on the *minimum spanning tree* heuristic. We also used a variant of this domain where the cost of moving does not change after adding dirt, in which case the problem is basically TSP.

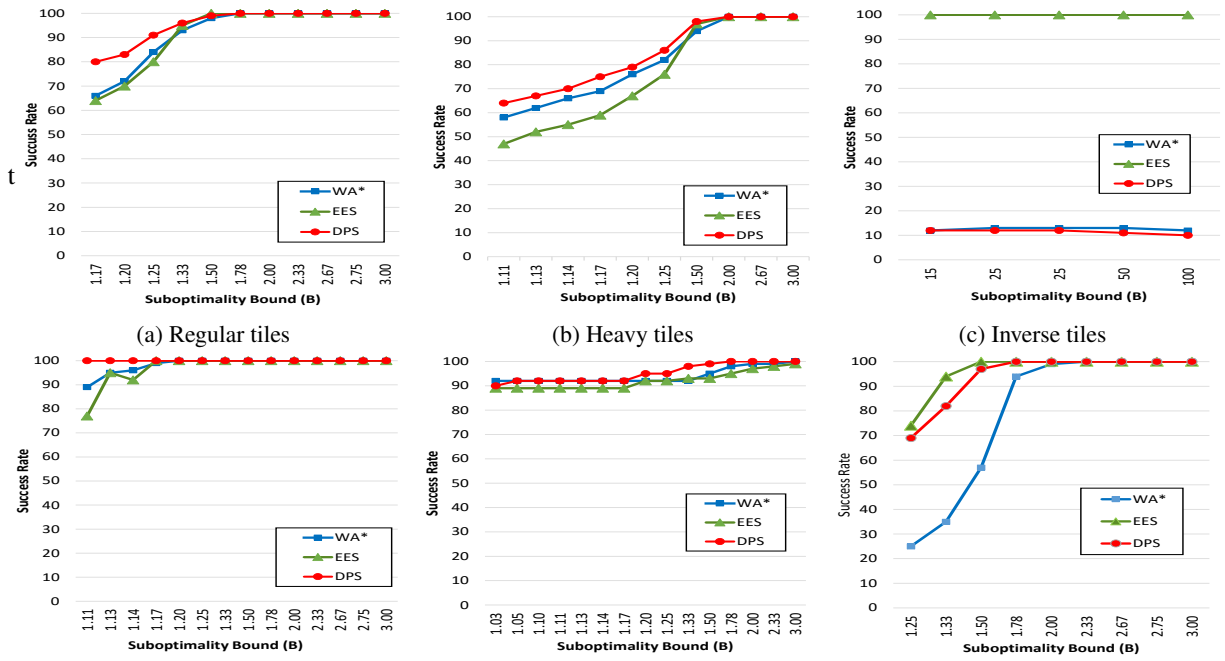
The Dock Robot Domain This domain was also introduced by Thayer and Ruml (2011) and is inspired by Ghallab et al. (2004) and by the depots domain from the IPC. A robot is tasked to move containers from one location to another. Similar to the blocksworld domain, containers are stacked on different piles and only the topmost container on a pile may be accessed at any time. Stacking and unstacking a container Z is done by a crane and costs 0.05 times the height of the related pile that has Z. To move a container between piles, the robot needs to also load it, which costs 0.1, drive to the desired pile, which costs the distance between the piles, and finally the robot unloads the container, again incurring a cost of 0.05. See Thayer and Ruml (2011) for the details of how h and d were computed.

6 Experimental Results

WA*, DPS and EES are all best-first searches that use OPEN lists. Since our domains are exponential, some of the problem instances could not be solved under our memory limitations. Therefore, we limited the number of nodes that could be generated before the problem is solved to 5 million. If the problem was not solved under this limit, it was marked as failed. The percentage of problems solved before reaching this limit is referred to as the "success rate".

Figure 1 shows the success rate over 100 random instances (y-axis) for representative domains which include: The regular, heavy and inverse variants of the 15-puzzle, the regular 101-pancake puzzle, the heavy 16-pancake puzzle, and the dock robot domain. The x-axis gives the desired suboptimality bound (B). These results indicate the following trends: (1) DPS outperformed EES in the four leftmost domains (Regular and heavy 15-puzzle, 101-pancake and dock robot). (2) DPS was slightly inferior in one domain (heavy pancake) and (3) DPS is losing completely in another domain (inverse tiles). WA* is losing in all these domains and we focus below on DPS and EES. When we analyzed running times and number of nodes expanded we saw a clear trend that usually, whenever there was a win in the success rate there was a win in the nodes count and running time too. We show representative results for some of our domains below. The constant time per node was slightly larger for EES as can be seen in our CPU times reported below.

[7] The motivation is that to flip i pancakes you need a spatula that is big enough to hold the larger side of the flipped set of pancakes, or else it will topple. Operating a bigger spatula is more costly.



(a) Regular tiles (b) Heavy tiles (c) Inverse tiles

(d) 101 pancakes (e) Dock robot (f) 16 heavy pancakes

Figure 1: Success rate (y-axis) for values of B (x-axis) for 6 domains. Vacuum cleaner had 100% success rate and is omitted.

		Nodes expanded			CPU Time in ms		
B	#Ins	WA*	EES	DPS	WA*	EES	DPS
101-pancake							
1.11	68	27,612	26,688	**912**	20,027	24,054	**695**
1.14	89	11,078	13,062	**332**	8,383	11,336	**230**
1.25	100	1,399	626	**169**	952	494	**110**
1.50	100	181	**147**	148	115	100	**98**
Dock robot							
1.10	89	73,753	88,027	**56,856**	4,114	5,282	**2,558**
1.50	92	47,460	52,805	**23,189**	2,110	2,867	**824**
3.00	99	28,176	47,550	**16,451**	1,032	2,842	**543**

Table 3: 101 Pancakes (top). Dock Robot (Bottom).

Our aim in the remainder of this paper is to analyze the results of Figure 1 and better understand when each algorithm performs better. For this we provide a number of general rules on when we expect one of the algorithms to be better than the others.

6.1 The Strength of d and h

Rule 1: One of the main strengths of EES is that it uses both d and h and intelligently combines them together. But, in cases where $d = h$ such as unit-cost domains or when d and h are very close to each other EES loses its strength. Naturally, the weakness of EES relatively strengthens DPS when they compete.

To illustrate Rule 1, consider the results of the 101-pancake puzzle in Figure 1(d). Note that the largest reported optimal solver for this domain was for the 85-pancake puzzle (Lippi, Ernandes, and Felner 2016). DPS solved all instances of the 101-pancake even for $B = 1.11$ while EES could only solve 75% of the instances for $B = 1.11$. Table 3 (top) shows the average number nodes expanded and CPU time for the instances solved by all algorithms (The $\#Ins$ column shows the number of such instances). Again, the huge benefit of DPS is observed; it outperformed EES by a factor of 27. The poor performance of EES compared to DPS on the 101-pancakes, as well as in the 15-puzzle (Figure 1a) is expected, as these domains have unit edge costs and thus $d = h$ and EES loses much of its strength as explained by Rule 1.

A less intuitive result is that DPS outperforms EES in the non-unit edge cost dock robot domain (Figure 1e). Nodes expanded and CPU time are reported in Table 3 (bottom) and indicate an advantage of up to a factor of 3 for DPS over EES. A possible explanation is that in this domain the differences between action costs is not very large, and thus the difference between d and h is not large too. We discuss in Section 6.4 how this is related to the performance of DPS.

The results for the heavy tiles are mixed (Figure 1b). For $B \leq 2$ DPS is better but for $B > 2$ the problem is relatively easy and both DPS and EES had a success rate of 100%. However, EES was able to do so while expanding many fewer nodes. For example, for $B = 2$, 3, and 15, DPS expanded 1.7, 4.3, and 26 times more nodes than EES, respectively. This behavior – EES being faster for large values of B – can be explained as follows. For large B values almost all nodes will lead to a sufficient solution, and the only challenge is to find any solution as fast as possible. Thus, EES will converge to Speedy search (Wilt and Ruml 2014) (using d only) while DPS will converge to GBFS as h will be a dominating factor in the ud formula ($B \times f_{min} - g$ remains constant). In the heavy 15-puzzle it is clear that

		regular						heavy					
		Nodes expanded			CPU Time in ms			Nodes expanded			CPU Time in ms		
B	SR	WA*	EES	DPS	WA*	EES	DPS	WA*	EES	DPS	WA*	EES	DPS
1.11	100	610,229	**481,760**	489,647	3,935	3,529	**3,118**	2,440,032	**2,120,931**	2,253,327	22,904	**22,043**	22,827
1.14	100	505,922	**363,889**	388,107	3,400	2,757	**2,616**	2,109,272	**1,711,585**	1,916,903	18,431	**18,206**	18,318
1.17	100	439,055	**300,359**	328,699	2,991	2,211	**2,106**	1,889,898	**1,448,847**	1,700,742	16,444	**14,182**	16,079
1.25	100	272,375	**159,027**	197,225	1,771	**1,143**	1,251	1,294,335	**820,477**	1,131,472	10,714	**7,408**	10,603
1.50	100	77,162	**41,025**	62,200	445	**270**	366	464,565	**248,497**	400,121	3,703	**2,067**	3,545
2.00	100	30,584	**14,492**	25,552	176	**102**	165	185,202	**97,591**	169,742	1,370	**741**	1,380
2.33	100	22,468	**10,310**	20,434	117	**59**	118	145,459	**55,383**	138,602	1,056	**412**	1,136
3.00	100	21,000	**8,137**	18,714	115	**51**	132	148,189	**18,933**	144,171	1,213	**131**	1,169

Table 4: Average nodes expanded and runtime for the regular and heavy vacuum with 10 dirty spots

	Success rate diff (DPS-EES)					Expansion ratio (EES/DPS)				
Bound	0.0	0.5	1.0	1.5	2.0	0.0	0.5	1.0	1.5	2.0
1.10	0	1	43	-	-	3.2	14.1	26.4	-	-
1.17	0	0	6	57	93	5.6	10.5	135.8	81.8	59.4
1.20	0	0	0	37	83	3.1	7.6	49.6	79.5	180.2
1.33	0	0	0	1	21	1.0	1.4	9.1	43.4	80.5
1.50	0	0	0	0	0	1.0	1.0	1.1	4.6	11.0
2.00	0	0	0	0	0	1.0	1.0	1.0	1.1	1.2

Table 5: Results for 40-pancake with the GAP-X heuristic

Speedy search would be much faster as there is no value in considering h when searching for any solution.

The results for the heavy 16-pancake (Figure 1f), and especially for the inverse 15-puzzle (Figure 1c) are much clearer. In these non-unit edge cost domains, DPS is significantly inferior to EES. These are domains where d is very different from h and thus EES is especially strong.

6.2 The Impact of a Weaker Heuristic

Rule 2: DPS is robust across heuristics but EES suffers when weakening the heuristic.

To show this we performed additional experiments on the pancake puzzle. We chose this puzzle as it has an extremely accurate heuristic – GAP (Helmert 2010), and recent work showed a simple way to derive less accurate heuristics from GAP for scientific purposes, called the GAP-X family of heuristics (Holte et al. 2016). In GAP-X, we do not count the gaps involving any of the X smallest elements. For example, GAP-2 does not count the gaps involving elements 1 or 2. We further fine tuned the GAP-X family to also include "half gaps" as follows. In GAP-X we do not count gaps of any element $\leq X$ either from its left or its right. We define $GAP - X.5$ to include all gaps up to element $X - 1$ but only the GAP to the left of element X. For example, assume the 5-pancake puzzle where the elements appear in the following vector $[3, 1, 5, 2, 4]$. There are four gaps which occur on both sides of elements 1 and 2, and one to the right of 4. Therefore the GAP heuristic will return 5. GAP-1, will only accumulate the gaps that do not involve element 1 and return 3. Finally, GAP-1.5 will not count the gaps that involve element 1 and those that involve element 2 from its left. It will return 2 (i.e., only the gap $(2, 4)$ and the gap to the right of 4 will be counted.)

Table 5 shows results for the 40-pancake puzzle with GAP, GAP-0.5 ... GAP-2. The rows correspond to different values of B and columns correspond to the different values of X. In this domain, DPS always had a success rate of 100%, while for some of values of X EES had a much lower success rate. The columns under "Success rate diff (DPS-EES)" show the success rate of DPS minus the success rate of EES. The columns under "Expansion ratio (EES/DPS)" show the average number of nodes expanded by EES divided by the average number of nodes expanded by DPS for the instances solved by both DPS, EES, and WA*. For both "success rate diff" and "expansion ratio", higher numbers mean a bigger advantage to DPS over EES.

These results confirm Rule 2. When weakening GAP, EES solves fewer instances and expands more nodes than DPS. For example, using GAP, EES solves all instances for $B = 1.20$ but with GAP-1.5 and GAP-2, EES solves 37% and 83% less instances than DPS, respectively. Similarly, with GAP, EES and DPS expand about the same number of nodes, but with GAP-1.5 EES expand 79 times more nodes than DPS. With GAP-2 the expansion ratio is 180.

6.3 The Accuracy of $\hat{h}$

Rule 3: EES is very strong when $\hat{h}$ is relatively accurate.

DPS may be weaker than EES even in unit edge cost domains. Table 4 shows the average number of nodes expanded and CPU time required to solve problem instances for the unit-edge cost Vaccum domain, where dirt does not weigh (left) and for the non-uniform edge cost variant, where picking up dirt makes the cleaner heavier (right). These results show that EES and DPS perform similarly on smaller values of B, but for larger values of B EES is better than DPS, even in the unit edge cost variant.

This is explained as follows. A key part of EES are $\hat{h}$ and $\hat{d}$ which are learned during the search, and are expected to become more accurate as the search progresses. Thayer and Ruml (2011) suggested to compute the learned heuristics of a node n by considering the heuristic errors observed *along the path from the start to* n.(This is called the path-based error correction method (Thayer and Ruml 2011)). We follow their method. Thus, in domains where the solution is relatively shallow, $\hat{h}$ will be uninformed, basing its learning on a small sample. In the vacuum world, solution depths are significantly larger than in all other domains, averaging over 800 steps (compared to 52 in the 15-puzzle). Moreover, the heuristic error in the vacuum world is expected to be similar throughout the search, as the obstacles in this domain are

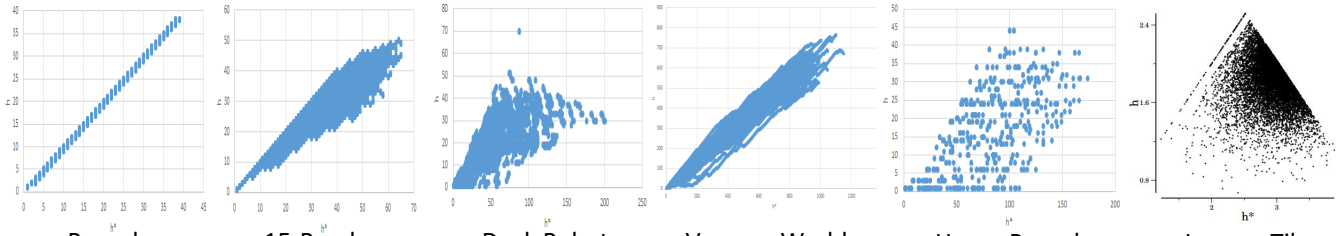


Figure 2: The h^*-values (x-axis) plotted against the h-values (y-axis) of states of 6 domains (Right plot taken from Thayer et al. (2012)).

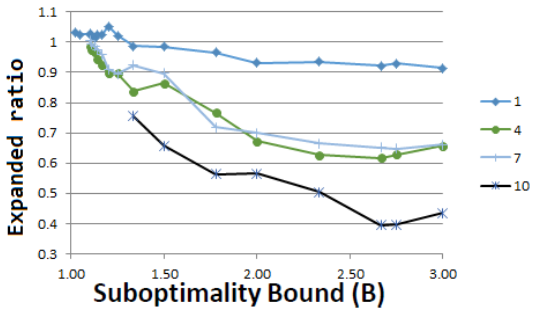


Figure 3: Varying the dirt for the vacuum cleaner

uniformly distributed. Thus, it is reasonable to learn it and obtain accurate $\hat{h}$. This makes EES strong here despite the fact that this is a unit-cost domain.

To demonstrate that long paths strengthen $\hat{h}$ and EES, we varied the number of dirty locations that the cleaner needs to pick up. Less dirt means shallower solution depth. Consequently, we expect EES to perform worse as we decrease the amount of dirt. The results are shown in Figure 3. The x-axis corresponds to B and the y-axis gives the average ratio of nodes expanded by EES divided by the average number of those expanded by DPS (lower values correspond to better performance of EES). Different curves shows results for different amount of dirt. The results support our hypothesis: indeed, more dirt is better for EES. With low B values, both algorithm converge to A* and their results are similar.

6.4 Heuristic Error Models

Rule 4: *linear relative* h-to-h^* ratio strengthens DPS.

A question that remains not fully answered is: why does DPS perform well on dock robot, although it is a non-unit edge cost domain? We argued that EES is expected to be strong for problems with longer solutions. In our experiments, the average solution depth for a dock robot problem was 25 (see Table 2). This explains why EES was not very effective in this domain.

However, there is an additional reason. The motivation for using DPS is that it expands in every iteration the node that is most likely (higher potential) to be within the bound. However, this is true for domains and heuristic that exhibit the *linear relative heuristic error model* (Stern et al. 2014). Informally, this means that the error of the heuristic can be modeled as a random variable multiplied by the heuristic value. This error model will strengthen DPS but will weaken EES, as we expect nodes with large h-values to have a wider variance of h^* values and learning $\hat{h}$ is more problematic.

To examine which of the evaluated domains exhibits this heuristic error model, we plotted the heuristic value h against the optimal solution h^* for a large number of nodes. Figure 2 shows the resulting plot. The results match nicely with the observed performance of the algorithms. The regular (not heavy) pancake puzzle and the 15-puzzle exhibit a clear linear-relative h-to-h^* relation. The dock robot (even though it is a non-unit edge cost domain) still exhibits this relation: the h^* values become more distributed as the value of h gets larger. This explains why DPS works well in this domain. Both vacuum and heavy pancake do not show this h to h^* behavior. In both domains (more emphasized in the heavy pancake), the range of h-to-h^* values is more constant than linearly dependent on h. Thus, the estimates of DPS of which node is more likely to be below the bound is less accurate, and this weakens DPS. Indeed, even in the unit-edge cost variant of Vacuum DPS was outperformed by EES, and in the heavy pancake EES was the clear winner. Finally, it is easy to see that the inverse 15-puzzle, certainly the h-to-h^* is not linear. This correlates perfectly with DPS's poor performance in this domain.

Summary We provided a number of general rules. It is very hard to predict the net effect of these rules for a given domain. Nevertheless, based on our study we can summarize them as follows. DPS is stronger than EES when there is a linear h-to-h^* relation and when $d \approx h$. EES is stronger than DPS when d and h are different and when $\hat{h}$ is expected to be accurate, e.g., in domains with large solution depth. In addition, DPS is easier to implement as it only needs a single OPEN list while EES stores three open lists and needs to learn $\hat{d}$ and $\hat{h}$.

7 Conclusions

In this work we showed how to adapt a general class of BCS algorithms to solve BSS. In particular, we migrated PS, a BCS algorithm, to the BSS setting. The resulting algorithm, DPS, is shown to be very efficient on a variety of domains. Through an extensive experimental evaluation, we identified several general guidelines where DPS is expected to perform well compared to EES and vice versa.

8 Acknowledgements

The research was supported by the Israeli Science Foundation (ISF) under grant #417/13 to Ariel Felner.

References

Bulitko, V.; Björnsson, Y.; Sturtevant, N. R.; and Lawrence, R. 2011. Real-time heuristic search for pathfinding in video games. In *Applied Research in Artificial Intelligence for Computer Games*. Springer.

Burns, E. A.; Hatem, M.; Leighton, M. J.; and Ruml, W. 2012. Implementing fast heuristic search code. In *SoCS*.

Ebendt, R., and Drechsler, R. 2009. Weighted A* search - unifying view and application. *Artif. Intell.* 173(14):1310–1342.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated planning: theory & practice*. Elsevier.

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

Helmert, M. 2010. Landmark heuristics for the pancake problem. In *Third Annual Symposium on Combinatorial Search (SoCS)*.

Holte, R. C.; Felner, A.; Sharon, G.; and Sturtevant, N. R. 2016. Bidirectional search that is guaranteed to meet in the middle. In *Proceedings of the AAAI Conference on Artificial Intelligence*.

Lippi, M.; Ernandes, M.; and Felner, A. 2016. Optimally solving permutation sorting problems with efficient partial expansion bidirectional heuristic search. *Artificial Intelligence Communication*.

Pearl, J., and Kim, J. H. 1982. Studies in semi-admissible heuristics. *IEEE Trans. Pattern Anal. Mach. Intell.* 4(4):392–399.

Pohl, I. 1970. Heuristic search viewed as path finding in a graph. *Artificial Intelligence* 1(3-4):193–204.

Pohl, I. 1973. The avoidance of (relative) catastrophe, heuristic competence, genuine dynamic weighting and computational issues in heuristic problem solving. In *IJCAI*, 12–17.

Russell, S., and Norvig, P. 1995. *Artificial intelligence: a modern approach*. Prentice Hall.

Stern, R.; Felner, A.; van den Berg, J.; Puzis, R.; Shah, R.; and Goldberg, K. 2014. Potential-based bounded-cost search and anytime non-parametric A*. *Artificial Intelligence* 214:1–25.

Stern, R.; Puzis, R.; and Felner, A. 2011. Potential search: a bounded-cost search algorithm. In *ICAPS*.

Thayer, J. T., and Ruml, W. 2011. Bounded suboptimal search: A direct approach using inadmissible estimates. In *IJCAI*.

Thayer, J. T.; Stern, R.; Felner, A.; and Ruml, W. 2012. Faster bounded-cost search using inadmissible estimates. In *ICAPS*.

Valenzano, R. A.; Arfaee, S. J.; Thayer, J. T.; Stern, R.; and Sturtevant, N. R. 2013. Using alternative suboptimality bounds in heuristic search. In *ICAPS*.

Wilt, C. M., and Ruml, W. 2014. Speedy versus greedy search. In *Seventh Annual Symposium on Combinatorial Search*.