Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence

Forward Perimeter Search with Controlled Use of Memory

Thorsten Schütt, Robert Döbbelin, Alexander Reinefeld

Zuse Institute Berlin, Germany

Abstract

There are many hard shortest-path search problems that cannot be solved, because best-first search runs out of memory space and depth-first search runs out of time. We propose *Forward Perimeter Search (FPS)*, a heuristic search with controlled use of memory. It builds a perimeter around the root node and tests each perimeter node for a shortest path to the goal. The perimeter is adaptively extended towards the goal during the search process.

We show that FPS expands in random 24-puzzles 50% fewer nodes than BF-IDA* while requiring several orders of magnitude less memory.

Additionally, we present a hard problem instance of the 24-puzzle that needs at least 140 moves to solve; i.e. 26 more moves than the previously published hardest instance.

1 Introduction

Many heuristic search algorithms have been devised to find a shortest path in a graph. A* expands the fewest nodes, but its applicability is limited because it holds all expanded nodes in the main memory. IDA* in contrast, uses essentially no storage and is fast in terms of node expansions per second. However, it has four weaknesses: (1) Its iterative-deepening node expansion visits the same nodes multiple times, (2) it is not able to detect repeated nodes in a graph because it does not keep information on the visited nodes, (3) it uses a strict left-to-right traversal of the leaf nodes whereas A* maintains the frontier in a best-first order, and (4) it does not keep information from the preceeding iteration.

Breadth-first search with heuristic pruning has been proposed as a compromise between A* and IDA*. It needs less memory space than A* for storing the search front and is still able to detect repeated nodes in graphs, thereby solving weaknesses (2) and (3). Several variants have been proposed: breadth-first frontier search [Korf and Zhang, 2000], breadth-first heuristic search [Zhou and Hansen, 2004], and breadth-first iterative-deepening A* (BF-IDA*) [Zhou and Hansen, 2004].

Part of this work was supported by the EU project CONTRAIL 'Open Computing Infrastructures for Elastic Services'.

But the applicability of breadth-first search is limited to problems where the largest search front fits into the main memory. Parallel implementations of BF-IDA* [Schütt *et al.*, 2011] alleviate this problem by partitioning the search front over several computers and thereby utilizing the main memories of all parallel machines as a single aggregated node store. Although these algorithms have been shown to run efficiently on parallel systems with more than 7000 CPU cores, there exist large problems that cannot be solved with BF-IDA*.

Forward Perimeter Search (FPS) allows to steer the memory consumption within certain limits and it does not suffer from IDA*'s weaknesses (2) to (4). FPS first generates a set of nodes (the perimeter P) around the root node and then performs heuristic breadth-first searches to test whether any perimeter node $p \in P$ reaches a solution within a given threshold. When no solution has been found, the threshold is increased and the next iteration is begun. Before starting a new iteration, FPS checks whether a subtree of a perimeter node p exceeds a given size limit and, if this is the case, enlarges the perimeter at p. By this means, the perimeter is iteratively extended in the most promising direction, because the test subtrees grow in the direction of the expected goal. All perimeter nodes are kept in the main memory and their goal distance estimates are updated.

The adaptive decomposition of the search space allows FPS to solve large problems without exceeding the available memory. FPS' breadth-first expansion of the perimeter nodes avoids duplicate node expansions as far as possible (depending on the available memory) and its adaptively refined perimeter makes it likely to find a goal early in the last iteration. This perimeter refinement is the main reason for the 50% node savings with respect to BF-IDA* (Sec. 4).

FPS can be executed on massively parallel systems and on clusters. It provides two sources of parallelism. First, all perimeter nodes can be searched concurrently without incurring any communication overhead ('trivial parallelism'). Second, each perimeter node can be tested with the parallel BF-IDA* algorithm [Schütt *et al.*, 2011] presented in Sec. 3.4. In fact, all empirical results in Sec. 4 were obtained on compute clusters of various sizes.

This paper begins with a brief review of related work. Thereafter, we introduce FPS and present empirical results on the 24-puzzle and 17-pancake problem. Finally, we present an instance of the 24-puzzle that is more difficult to solve

than any previously published instance and we give upper and lower bounds (140 resp. 142 moves) on its solution length.

2 Related Work

The idea of perimeter search was invented almost twenty years ago. But in contrast to our approach, the previous algorithms [Dillenburg and Nelson, 1994; Manzini, 1995; Kaindl and Kainz, 1997] build a perimeter around the goal rather than the start node[1]. Hence, each newly expanded node must be checked against all perimeter nodes which only pays off in application domains with expensive operator costs.

Single-frontier bidirectional search [Felner *et al.*, 2010] also builds a search front around the goal node, but it dynamically switches the search direction between forward and backward search. Several jumping policies (highest branching degree, smallest h-value) have been evaluated. Further improvements can be achieved by combining this method with *multiple-goal pattern databases* [Felner and Ofek, 2007] which are seeded with the states' values that are abstracted from the perimeter nodes.

Fringe search [Björnsson *et al.*, 2005] stores the search frontier (fringe) in main memory. It searches the graph in an iterative-deepening fashion without re-expanding the nodes inside the fringe. The algorithm is beneficial when inaccurate heuristics would require many iterations, but it does not allow to control the memory consumption.

MREC [Sen and Bagchi, 1989] is a combination of A* and IDA*. It stores as much as possible of the explicit search graph in memory and searches the remaining nodes with IDA*. MREC is comparable to FPS, but instead of keeping the search space in memory, we only store a search frontier (the perimeter) and instead of using IDA* we use BF-IDA* to test the tip nodes. But more importantly, we incrementally extend the perimeter towards the goal with information from the previous iteration.

Bidirectional BF-IDA* [Barker and Korf, 2012] can find an optimal solution without performing the last (most costly) iteration in which the threshold is equal to the optimal solution cost. The potential node savings are high, but the algorithm does not allow to control the memory consumption.

3 Forward Perimeter Search (FPS)

We consider shortest path search in undirected graphs with non-negative edge costs. The graph is represented implicitly by a procedure for generating the successors of a node.

Forward Perimeter Search (FPS) has two phases: It first builds a perimeter around the start node s and then tests for each perimeter node whether it lies on a shortest path from s to the goal g. The testing can be done with a variety of search algorithms. We use BF-IDA*. Both phases are executed in an iterative-deepening manner. Before starting the next iteration with a deeper search depth, the perimeter is adjusted by taking information from the preceeding iteration into account. The perimeter nodes are sorted so that a goal is found early in the last iteration.

[1]We therefore named our algorithm *Forward* Perimeter Search.

Algorithm 1 Computing a perimeter with radius r

```
peri extend_perimeter(peri P;node s,p;int r){
  P = P \ {p};
  C = circle(p, r);
  foreach(node n ∈ C)
    if(d(s,n) == d(s,p) + r)
      P = P ∪ n;
  return P;
}
```

3.1 Perimeter

Let $sp(x, y)$ be the set of all shortest paths between x and y and let $d(x, y)$ be the length of the shortest path(s). A *perimeter* is a set of nodes around a start node s such that any shortest path from s to a goal node g passes through at least one node of the perimeter.

Definition 1 (Perimeter). *A set of nodes P in a graph $G = (V, E)$ is a perimeter around s for the shortest paths between $s, g \in V$, if $\forall$ path $\in sp(s, g)$: path $\cap P \neq \emptyset$.*

A perimeter P can be incrementally built from a single start node $P = \{s\}$. The function `extend_perimeter` (Alg. 1) does this by removing a node p from a perimeter P and inserting all descendants v into P which are r steps away from p and also $d(s, p) + r$ steps away from the start s. The distance $d(s, v)$ can be determined with a short backward BF-IDA* search from v to the start node s. It ensures that backwards lying nodes (dashed line in Fig. 1) will not be inserted into P.

This scheme will be used to incrementally enlarge the perimeter towards the expected goal. Before describing the FPS algorithm with an adaptive radius (Sec. 3.3), we present a simpler version which uses a perimeter with fixed radius.

3.2 FPS with Fixed Radius

The simple version of the FPS algorithm builds a perimeter with a fixed radius. This is done by expanding all nodes that are r steps away from the start s as shown in the function `extend_perimeter` in Alg. 1. When the perimeter P has been built, the search process continues in an iterative-deepening fashion. In each iteration, it tests for each node $p \in P$ whether there is a path of length *thresh* $- r$ from p to the goal g. If yes, we terminate the search with a shortest path of length *thresh*. If not, we increase *thresh* by the least cost δ of all paths that exceeded *thresh* in the last iteration.

Note that all perimeter nodes are tested in each iteration except the last. In the last iteration, the search is stopped as soon as a solution is found. We therefore sort all nodes $p \in P$ so that the most promising ones will be tested first. We experimented with several sorting schemes (Sec. 4.1) and found that simple schemes such as *longest path first* are already close to the optimum.

The testing can be done with a wide variety of node expansion strategies: breadth-first, best-first, depth-first or breadth-first frontier search. We used the parallel variant [Schütt *et al.*, 2011] of breadth-first iterative deepening A* (BF-IDA*) [Zhou and Hansen, 2004]. It is efficient and it never revisits a node in the same iteration.

Algorithm 2 Forward Perimeter Search (FPS)

```
int FPS(node s,goal; radius r) {
  int thresh = h(s);
  peri P = {s};
  P = extend_perimeter(P,s,s,r);
  while(true) {
    sort(P);
    foreach(node p ∈ P)
      if (test(p, goal, thresh − d(s, p)))
        return thresh;
    foreach (node p ∈ P)
      if(p.tree_size > limit)
        P = extend_perimeter(P, s, p, r);
    thresh = thresh + δ;
  }
}
```

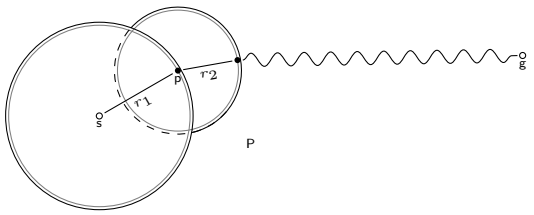

Figure 1: Perimeter adaptation in FPS.

The size of the radius is of crucial importance for the performance and the memory consumption of FPS. The memory needed to store the perimeter nodes must be carefully balanced with the memory requirements of the test function that is started at the perimeter nodes. As we will see in Sec. 4.1 larger radii contain more perimeters nodes and need more memory space, but their smaller subtrees are beneficial for quickly finding a solution in the last iteration. Smaller radii, on the other hand, spawn larger subtrees at the perimeter nodes, which allows to eliminate more duplicates.

3.3 FPS with Adaptive Radius

It is difficult to select a suitable fixed radius without knowing the properties of the search graph. The FPS variant with an adaptive radius uses information from the previous iteration to adaptively extend the perimeter. It starts with a small perimeter and extends it (using Alg. 1) at those perimeter nodes that spawned the biggest subtrees in the last iteration. This is done with the expectancy that these nodes are more likely to lie on a shortest path and, even more important, that the final iteration is speeded up by searching a small subtree (if the sorting was good). This adaptive perimeter refinement eliminates the major weakness of BF-IDA*, namely the large amount of node expansions in the last iteration.

Fig. 1 illustrates a scenario where an initial perimeter with radius r_1 was built and later extended at node p by another radius r_2 around p. Note that the nodes on the dashed line should not be included in the perimeter because they are redundant and the property in Def. 1 holds without them.

Alg. 2 shows the pseudo-code of FPS with adaptive radius.

Algorithm 3 The BF-IDA* `test` function used in Fig. 2.

```
void mapper(node n, set<node> pred) {
  foreach(succ in get_adjacent_nodes(n))
    if(!pred.contains(succ))
      if(g + 1 + h(succ) <= thresh) //pruning
        emit(succ, n);
}

void reducer(node n,list<node> predlst){
  set<node> preds = {};
  foreach(p in predlst)
    preds.add(p); //merge predecessors
  solved = solved ∨ pos == goal;
  emit(n, preds);
}

bool test(node s, goal; int thresh) {
  frontier = [(s, {})];
  g = 0; solved = false;
  while (!solved ∧ frontier.size() != 0) {
    intermediate = map(frontier, mapper);
    frontier= reduce(intermediate,reducer);
    g++;
  }
  return solved;
}
```

The search is started by building an initial perimeter with radius r. FPS then performs three steps in a loop:

1. It orders the perimeter nodes $p \in P$ according to the information gathered in the previous iteration so that the most promising nodes are tested first.
2. It iterates over all perimeter nodes and tests whether any of them lie on a shortest path between s and g within *thresh*. If this is true, the search is terminated with *thresh* as the solution length.
3. It checks for all subtrees spawned at perimeter nodes p whether they surpassed the given memory $limit$. If this is the case, the radius is enlarged at this node. The size of the new radius depends on the branching factor of the graph and thus is domain dependent.

In our implementation, we store for each perimeter node: the distance to the start node, the maximum path length towards the goal, the number of expanded nodes and the widest search front in the last iteration.

3.4 Testing the Perimeter Nodes with BF-IDA*

The `test` function in Alg. 2 spawns the subtrees from the perimeter nodes. Many different node expansion strategies can be used for the testing. We used the parallel BF-IDA* algorithm [Schütt *et al.*, 2011] which is based on [Zhou and Hansen, 2004]. It expands all nodes of a search frontier in parallel and eliminates duplicates on-the-fly.

The parallel BF-IDA* uses the MapReduce framework [Dean and Ghemawat, 2008] for orchestrating the concurrent process execution. Alg. 3 shows the program code of `test`. The search space is partitioned among all processes. In each step, the map processes apply the `mapper` function to all

nodes in the current frontier. For each node the mapper expands the successors and emits them to the next stage. Backward moves are eliminated by keeping track of the nodes' predecessors.

A global function is then used to assign the generated nodes to the reduce processes. The function (partially) sorts the nodes so that duplicates are assigned to the same reduce process. This is done with a hash function, which also provides a good load balancing over all reduce processes.

For each unique node found in its local dataset, the `reducer` merges the predecessors into a single set and emits the `node` with the joined set of predecessors. This implements the *delayed duplicate elimination* described in [Korf and Schultze, 2005; Zhou and Hansen, 2006]. The output of the `reducer` is fed as input into the next map phase until there are no more data pairs to process or a solution is found.

The described algorithm is efficient and simple to implement. The MapReduce framework orchestrates the concurrent `mapper` and `reducer` processes which iteratively expand the graph without visiting duplicates.

3.5 Reconstructing the Solution Path

At the end of the search, FPS returns only the cost of the shortest path but not the path itself. The path can be determined in two steps. First we compute the shortest path from s to the perimeter node p. This is easy because of the shallow search. Second, we determine the path from p to g which is done by recursively applying FPS. This is easier than the original problem since we know that the goal is $thresh - d(s, p)$ moves away from p. Hence the path can determined with a direct, i.e. non-iterative, FPS search.

4 Results

We first present empirical results on the 24-puzzle and thereafter on the 17-pancake problem. FPS was run on a cluster with 32 compute nodes, each of them with 2 quad-core AMD Opteron processors and 8 GB of main memory. BF-IDA* needed for the same problem instances a much larger system with more main memory. For the hardest problem we used 256 nodes, each equipped with 2 quad-core Intel Xeon processors and 48 GB of main memory. As a heuristic estimate function, we used the same 6-6-6-6 pattern database (PDB) with mirroring as in [Korf and Felner, 2002].

For a quick overview, Table 1 lists the node expansions and memory consumption on Korf's hardest problem instance of the 24-puzzle [Korf and Felner, 2002]. The performance of IDA* and BF-IDA* is given as a reference. As expected, IDA* expands the most nodes and requires the smallest memory space. BF-IDA* expands only one fourth of the IDA* nodes, but it keeps 50 billion nodes in the main memory, which is the widest search front in the final iteration. Note that it is not feasible to solve much larger problems with BF-IDA* because of its excessive memory requirements.

FPS outperforms BF-IDA* in two ways: It expands only approximately half of the nodes and, even more important, it needs far less memory—up to three orders of magnitude in this example. In the first set of experiments (lines 3-7) we used perimeters with fixed radii $r = 4 \ldots 18$. In the second set of experiments, we used FPS with an adaptively expanded perimeter. Here, FPS decides after each iteration for each perimeter node whether it should be extended so that the given memory limit is not overrun. The memory limits are given in terms of nodes, i.e. $1 \cdot 10^6$, $2 \cdot 10^6$, $35 \cdot 10^6$ and $188 \cdot 10^6$ nodes respectively. We extended the radius by at least $r \geq 3$ moves, depending on how much the limit was overshot in the previous iteration.

	node expansions	memory [nodes]
IDA*	4,156,099,168,506	113
BF-IDA*	1,067,321,687,213	50,675,640,000
FPS, r=4	423,306,411,815	5,922,529,960
FPS, r=6	428,072,054,940	2,876,547,362
FPS, r=10	564,996,269,605	1,220,873,196
FPS, r=16	647,863,040,082	503,869,879
FPS, r=18	671,310,216,245	257,590,848
FPS, $1.88 \cdot 10^8$	404,811,541,671	1,437,995,218
FPS, $3.5 \cdot 10^7$	452,935,148,947	1,078,733,091
FPS, $1.7 \cdot 10^7$	486,941,686,873	457,659,207
FPS, $2 \cdot 10^6$	619,262,051,017	112,469,403
FPS, $1 \cdot 10^6$	652,659,857,757	54,618,898

Table 1: Korf's hardest 24-puzzle (#50, 113 moves).

Most impressive is that fact that FPS with adaptive radius was able to solve the given problem instance on a single computer with only 8 gigabytes of main memory. BF-IDA*, in contrast, needed 256 compute nodes with several terabytes of main memory—and still expanded more nodes.

4.1 Fixed Radius

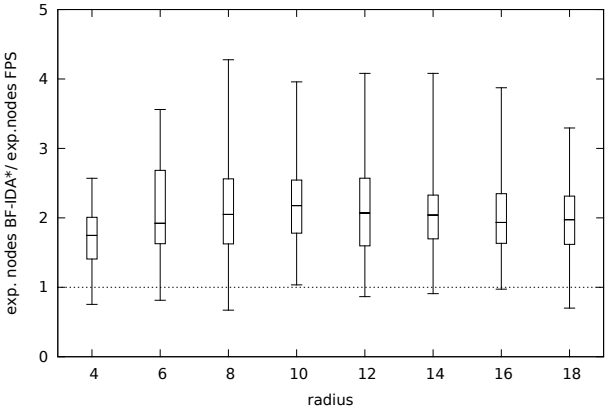

Figure 2: FPS node expansions relative to BF-IDA* for different radii on 40 instances of the 24-puzzle.

Fig. 2 compares the performance of FPS with various fixed radii to BF-IDA*. We could only run the first 40 instances of Korf's random set, because the search trees spawned by FPS with fixed radii were too large and it ran out of memory on the largest 10 instances. The graphs indicate, that FPS expands for all radii on the average only half of the BF-IDA* nodes. Note the stable improvement over all radii.

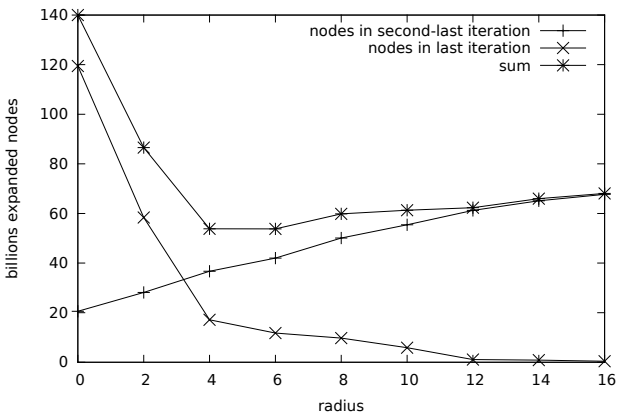

Figure 3: FPS node expansions for various radii (puzzle #17).

Fig. 3 illustrates that increasing the radius r yields dramatic node savings in the last iteration ($\times$ ticks), especially for $r \leq 4$. But unfortunately the node savings must be paid for by extra expansions in the second to last iteration ($+$ ticks). This is because larger radii contain more perimeter nodes which spawn overlapping subtrees in the testing. The perimeter nodes are tested separately, and hence duplicates cannot be eliminated. In the end these two effects compensate each other (see the $*$ ticks) and radii between 4 and 6 seem to be optimal. While this result was obtained from only a single problem instance (puzzle #17), it is in accordance with the larger test set shown in Fig. 4.

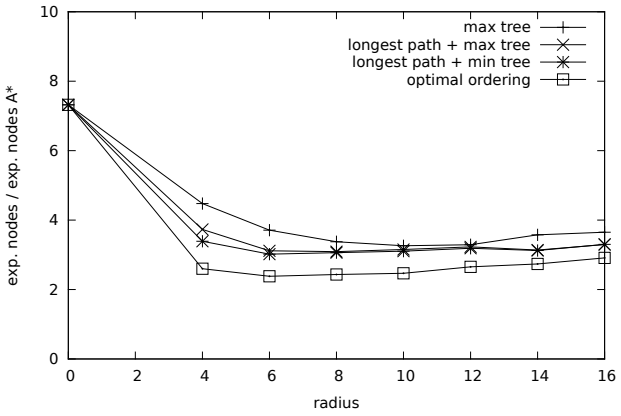

Figure 4: FPS node expansions relative to A* for different radii (x-axis) and different perimeter node sortings (diff. curves) on 40 instances of the 24-puzzle.

Fig. 4 shows the effect of sorting the perimeter nodes. The sort function uses information from the previous threshold:

- *max tree* sorts the perimeter nodes in decreasing order of their subtree sizes of the previous iteration.
- *longest path + max tree* favors the perimeter node with the longest path (i.e. max g). In case of ties, it takes the one with the bigger subtree.
- *longest path + min tree* is the same as above, but takes the smaller subtree in case of ties.
- *optimal ordering* shows the theoretical optimum that cannot be surpassed. It is achieved when the perimeter node leading to the goal is selected first.

Interestingly, all heuristics are close to the optimum. This indicates that further refinements probably do not pay off. We used *longest path + min tree* in all following experiments.

It can also be concluded from Fig. 4 that FPS expands 3.5 to 4 times more nodes than A*—but with much lower memory requirements. We obtained the A* performance by counting the nodes of BF-IDA* in the pre-final iteration, i.e. all nodes with $f \leq f^* - \delta$. These nodes are in A*'s Closed list just before A* finds a goal in the optimal case.

Compared to BF-IDA*, FPS saves almost half of the node expansions—again with less memory space. The leftmost tick at radius 0 shows the performance of BF-IDA*, which is equivalent to FPS with $r = 0$. IDA*, which is not plotted in the figure, expands approx. 32 times more nodes.

4.2 Adaptive Radius

id	d	IDA*	BF-IDA*	FPS	r_1	r_2
38	96	38,173,507	58,097,633	58,097,633	1.0	0.7
40	82	65,099,578	26,320,497	26,320,497	1.0	2.5
25	81	292,174,444	127,949,696	127,949,696	1.0	2.3
32	97	428,222,507	399,045,498	359,856,263	1.1	1.2
44	93	867,106,238	181,555,996	181,555,996	1.0	4.8
37	100	1,496,759,944	1,646,715,005	791,581,404	2.1	1.9
30	92	1,634,941,420	661,835,606	208,712,943	3.2	7.8
13	101	1,959,833,487	1,979,587,555	873,708,021	2.3	2.2
1	95	2,031,102,635	1,059,622,872	726,851,395	1.5	2.8
28	98	2,258,006,870	450,493,295	252,154,079	1.8	9.0
36	90	2,582,008,940	603,580,192	931,547,680	0.6	2.8
5	100	2,899,007,625	1,859,102,197	656,266,607	2.8	4.4
22	95	3,592,980,531	581,539,254	345,150,484	1.7	10.4
16	96	3,803,445,934	1,783,144,872	815,385,915	2.2	4.7
29	88	4,787,505,637	1,090,385,785	1,215,237,665	0.9	3.9
4	98	10,991,471,966	5,154,861,019	2,636,598,392	2.0	4.2
26	105	12,397,787,391	6,039,700,647	1,774,851,940	3.4	7.0
3	97	21,148,144,928	4,805,007,493	3,342,146,581	1.4	6.3
31	99	26,200,330,686	7,785,405,374	3,300,963,647	2.4	7.9
41	106	26,998,190,480	8,064,453,928	7,515,143,103	1.1	3.6
47	92	30,443,173,162	4,385,270,986	2,560,742,525	1.7	11.9
27	99	53,444,360,033	7,884,559,441	3,766,782,211	2.1	14.2
43	104	55,147,320,204	8,816,151,498	3,691,192,969	2.4	14.9
46	100	65,675,717,510	21,674,806,323	8,216,215,161	2.6	8.0
45	101	79,148,491,306	17,068,061,084	5,920,247,048	2.9	13.4
6	101	103,460,814,368	9,810,208,759	4,680,653,771	2.1	22.1
7	104	106,321,592,792	27,686,193,468	11,176,127,231	2.5	9.5
49	100	108,197,305,702	11,220,738,849	3,853,356,482	2.9	28.1
35	98	116,131,234,743	23,049,423,391	10,672,528,952	2.2	10.9
8	108	116,202,273,788	29,575,219,906	8,678,999,139	3.4	13.4
39	104	161,211,472,633	34,198,605,172	34,631,205,598	1.0	4.7
23	104	171,498,441,076	54,281,904,788	21,548,668,016	2.5	8.0
15	103	173,999,717,809	52,178,879,610	32,003,810,688	1.6	5.4
2	96	211,884,984,525	40,161,477,151	18,918,010,988	2.1	11.2
19	106	218,284,544,233	22,761,173,348	10,522,016,442	2.2	20.7
42	108	245,852,754,920	37,492,323,962	14,077,400,074	2.7	17.5
20	92	312,016,177,684	20,689,215,063	17,749,626,017	1.2	17.6
24	107	357,290,691,483	38,272,741,957	15,469,908,701	2.5	23.1
17	109	367,150,048,758	143,972,316,747	66,065,410,824	2.2	5.6
34	102	481,039,271,661	59,225,710,222	25,435,856,454	2.3	18.9
48	107	555,085,543,507	58,365,224,981	28,230,656,080	2.1	19.7
12	109	624,413,663,951	76,476,143,041	39,642,016,410	1.9	15.8
21	103	724,024,589,335	98,083,647,769	54,251,101,992	1.8	13.3
18	110	987,725,030,433	126,470,260,027	49,378,654,583	2.6	20.0
33	106	1,062,250,612,558	134,103,676,989	79,801,410,332	1.7	13.3
14	111	1,283,051,362,385	312,885,453,572	149,676,413,245	2.1	8.6
10	114	1,519,052,821,943	525,907,193,133	229,545,788,925	2.3	6.6
11	106	1,654,042,891,186	309,253,017,124	212,671,847,577	1.5	7.8
9	113	1,818,005,616,606	132,599,245,368	54,077,256,435	2.5	33.6
50	113	4,156,099,168,506	1,067,321,687,213	619,262,051,017	1.7	6.7
average		360,892,479,671	71,004,578,707	37,246,320,717	1.99	10.29

Table 2: Node expansions on Korf's fifty 24-puzzle instances with a 6-6-6-6 PDB. (id: Korf's Id, d: depth, r_1: BF-IDA*/FPS, r_2: IDA*/FPS)

Table 2 lists the performance of IDA*, BF-IDA* and FPS on the fifty random puzzles. We used the same 6-6-6-6 PDB with mirroring as a heuristic function. The perimeter was extended when the widest search front in a subtree exceeded $2 \cdot 10^6$ nodes.

The two rightmost columns r_1 and r_2 show the relative overhead of BF-IDA* and IDA* compared to FPS. FPS outperforms IDA* on the average by a factor of 10 and BF-IDA* by a factor of 2 in terms of node expansions. The benefits are more pronounced in the harder problem instances.

More important than the node savings are FPS' lower memory requirements. It allowed us to run all fifty instances on a single compute node with 8 GB of main memory whereas BF-IDA* needed a compute cluster with more than a terabyte to store the widest search front.

4.3 17-Pancake Problem

	exp. nodes	widest front
IDA*	137,148,572,155	15
BF-IDA*	478,979,227	97,629,006
FPS, r=2	227,904,600	13,398,750
FPS, $5 \cdot 10^6$	375,255,854	11,095,517

Table 3: Results on the 17-pancake problem. Average of ten random instances.

As a second benchmark we used the 17-pancake problem to evaluate the performance of FPS. An N-pancake problem is a stack of N pancakes of different size [Dweighter, 1975]. To solve the problem, the pancakes must be sorted by size. The only available operation is to reverse the order of a subset of pancakes at the top of the stack. Thus, a state has $N-1$ successors.

We generated ten random instances with an average solution length of 15.2 and performed a shortest path search. A PDB of the 8 topmost pancakes was used to guide the search. The results are summarized in Table 3. Compared to BF-IDA*, FPS reduces the number of node expansions by a factor of 2. Additionally the widest front and thus the memory consumption is reduced by a factor of 7.

5 A Hard Problem of the 24-Puzzle

Korf's fifty random instances of the 24-puzzle [Korf and Schultze, 2005] are often used as a benchmark for assessing the performance of search algorithms. Unfortunately, this set does not contain any really hard problem. As is known from the 15-puzzle, the distribution of the solution lengths of all puzzle instances gives a bell curve and hard problems are therefore unlikely to occur in a random set. Note that the hardest puzzle in Korf's set requires only 114 moves while [Karlemo *et al.*, 2000] presented a much higher upper bound of 210.

We constructed one problem instance of the 24-puzzle that is especially hard to solve (Fig. 5). Starting with the sorted puzzle on the left side in Fig. 5, we embed the hardest 15-puzzle (80 moves) into the upper left corner. We then compute the worst-case configuration for the lower fringe with

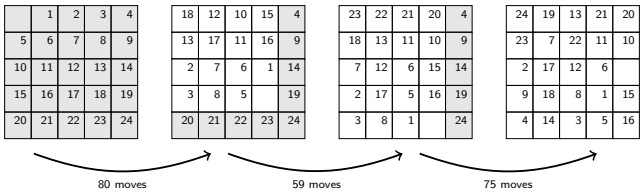

Figure 5: Constructing a hard 24-puzzle instance.

a breadth-first search. Thereafter, the same is done with the right fringe. Summing up the solution lengths gives an upper bound on the moves needed to solve this particular instance.

A direct solution of the hard problem in Fig. 5 is currently not possible. However, we can give a lower (140) and an upper bound (142) which are only two moves apart. The lower bound was computed by running FPS up to threshold 138 without finding a solution. This took three months.

thresh	6-6-6-6 PDB	8-8-8 PDB
126	399,633,789	51,115,210
128	3,468,558,764	457,928,595
130	29,048,297,692	3,986,628,500
132	393,504,563,894	33,417,370,606
134	3,996,860,914,262	457,717,114,294
136	-	4,499,126,967,518
138	-	42,854,920,933,846
140	-	?

Table 4: Trying to solve the hard problem with FPS.

Table 4 shows the node expansions of FPS with a memory limit of $5 \cdot 10^8$. In our experiments, we found the standard 6-6-6-6 PDB to be insufficient for solving this problem instance. Hence, we built a more powerful 8-8-8 PDB [Döbbelin *et al.*, 2013] which requires 122 GB memory space compared to 0.5 GB for the 6-6-6-6 PDB. Table 4 shows that the 8-8-8 PDB expands one order of magnitude fewer nodes. Even so, we were only able to run FPS up to threshold 138 without finding a solution[2].

We computed an upper bound by recursively calling FPS on the most promising perimeter nodes as illustrated in Fig. 6. From the first perimeter p_0, we picked a position t at distance 8 which looked promising according to the node ordering (4,499,126,967,518 node expansions). Since we were not able to compute the shortest path between t and g either, we again approximated the distance with the same approach (4,861,328,174,120 node expansions). We picked u from the perimeter p_1 and found an optimal path of length 119 from u to g (59,165,019,511 node expansions). Hence, the upper bound from s to g is $8 + 15 + 119 = 142$. This is only two moves longer than the lower bound 140 and we conclude that the optimal solution has either 140 or 142 moves which is $\geq$ 26 moves longer than the hardest instance in Korf's random set.

[2]IDA* could not be used either, because it expands approx. 10x more nodes. This extra effort is not compensated by IDA*'s faster node handling.

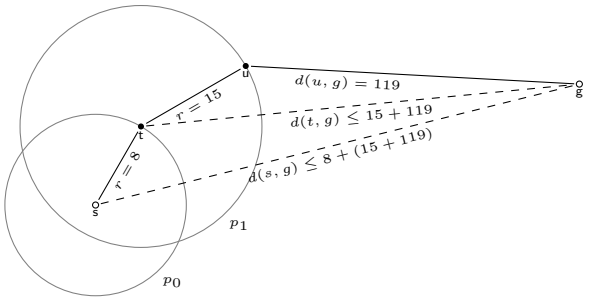

Figure 6: Computing the upper bound 142 with FPS.

6 Conclusion

We presented a heuristic search algorithm (FPS) that expands fewer nodes and requires several orders of magnitude less memory space than BF-IDA* on the 24-puzzle. This is possible by using information from the previous iteration to expand the perimeter in the direction of the expected goal. With its reduced memory consumption, FPS can be used to solve very large problems. As an example we solved the hardest 24-puzzle of the standard random benchmark set with 8 CPU cores and just 8 GB of memory—an instance for which BF-IDA* needs several terabytes.

FPS is flexible in two ways: It can dynamically trade the memory used for the perimeter with memory used by the test function, and it provides a template which can be used with various test functions. In the future we will experiment with better informed pattern databases [Döbbelin *et al.*, 2013] to further reduce the search effort of the test function.

We additionally presented a very hard problem instance of the 24-puzzle that can be used as a challenging benchmark for heuristic search algorithms. We applied FPS to this problem and established a lower and upper bound on the solution length which are only two moves apart.

References

[Barker and Korf, 2012] Joseph K. Barker and Richard E. Korf. Solving Peg Solitaire with Bidirectional BFIDA*. In *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence*, pages 420–426, 2012.

[Björnsson *et al.*, 2005] Yngvi Björnsson, Markus Enzenberger, Robert C. Holte, and Jonathan Schaeffer. Fringe Search: Beating A* at Pathfinding on Game Maps. In *Proceedings of the 2005 IEEE Symposium on Computational Intelligence and Games (CIG05)*, 2005.

[Dean and Ghemawat, 2008] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, pages 1–13, 2008.

[Dillenburg and Nelson, 1994] John F. Dillenburg and Peter C. Nelson. Perimeter Search. *Artificial Intelligence*, 65(1):165–178, January 1994.

[Döbbelin *et al.*, 2013] Robert Döbbelin, Thorsten Schütt, and Alexander Reinefeld. Building Large Compressed PDBs for the Sliding Tile Puzzle. Technical Report 13-21, Zuse Institute Berlin, 2013.

[Dweighter, 1975] Harry Dweighter. Problem E2569. *American Mathematical Monthly*, 82:1010, 1975.

[Felner and Ofek, 2007] Ariel Felner and Nir Ofek. Combining Perimeter Search and Pattern Database Abstractions. *SARA*, pages 155–168, 2007.

[Felner *et al.*, 2010] Ariel Felner, Carsten Moldenhauer, Nathan Sturtevant, and Jonathan Schaeffer. Single-Frontier Bidirectional Search. In *Third Annual Symposium on Combinatorial Search*, 2010.

[Kaindl and Kainz, 1997] Hermann Kaindl and Gerhard Kainz. Bidirectional Heuristic Search Reconsidered. *Journal of Artificial Intelligence Research*, 7:283–317, 1997.

[Karlemo *et al.*, 2000] Filip R. W. Karlemo, Patric R. J. Östergård, Tellabs Oy, and Patric R. J. On sliding block puzzles. *Journal of Combinatorial Mathematics and Combinatorial Computing*, 2000.

[Korf and Felner, 2002] Richard E. Korf and Ariel Felner. Disjoint pattern database heuristics. *Artificial Intelligence*, 134(1-2):9–22, January 2002.

[Korf and Schultze, 2005] Richard E. Korf and Peter Schultze. Large-scale Parallel Breadth-First Search. In *Proceedings of the National Conference on Artificial Intelligence*, volume 20, pages 1380–1385. AAAI Press / The MIT Press, 2005.

[Korf and Zhang, 2000] Richard E. Korf and Weixiong Zhang. Divide-and-Conquer Frontier Search Applied to Optimal Sequence Alignment. In *Proceedings of the National Conference on Artificial Intelligence*, pages 910–916. AAAI Press / The MIT Press, 2000.

[Manzini, 1995] Giovanni Manzini. BIDA: An Improved Perimeter Search Algorithm. *Artificial Intelligence*, 75(2):347–360, 1995.

[Schütt *et al.*, 2011] Thorsten Schütt, Alexander Reinefeld, and Robert Maier. MR-search: massively parallel heuristic search. *Concurrency and Computation: Practice and Experience*, August 2011.

[Sen and Bagchi, 1989] Anup K. Sen and Amitava Bagchi. Fast Recursive Formulations for Best-First Search That Allow Controlled Use of Memory. *IJCAI*, pages 297–302, 1989.

[Zhou and Hansen, 2004] Rong Zhou and Eric A. Hansen. Structured duplicate detection in external-memory graph search. In *Proceedings of the National Conference on Artificial Intelligence*, pages 683–689. AAAI Press / The MIT Press, 2004.

[Zhou and Hansen, 2006] Rong Zhou and Eric A. Hansen. Breadth-first heuristic search. *Artificial Intelligence*, 170(4-5):385–408, April 2006.