Anytime Pack Heuristic Search

Satya Gautam Vadlamudi[1], Sandip Aine[2], and Partha Pratim Chakrabarti[1]

[1] Indian Institute of Technology Kharagpur, Kharagpur, WB 721302, India
`{satya,ppchak}@cse.iitkgp.ernet.in`

[2] Indraprastha Institute of Information Technology, New Delhi, DL 110020, India
`sandip@iiitd.ac.in`

Abstract. Heuristic search is a fundamental problem solving technique in artificial intelligence. In this paper, we propose an anytime heuristic search algorithm called Anytime Pack Search (APS) which helps in solving hard combinatorial search problems efficiently. It expands nodes of a search graph in a localized best-first manner so as to converge towards good quality solutions at regular intervals. APS is complete on bounded graphs and guarantees termination with an optimal solution. Experimental results on the sliding-tile puzzle problem, the traveling salesman problem, and the single-machine scheduling problem show that APS significantly outperforms some of the state-of-the-art anytime algorithms.

Keywords: problem solving, heuristic search, anytime algorithms.

1 Introduction

Design of efficient anytime algorithms is one of the key aspects of time limited search based reasoning [2]. Heuristic search based methods are widely used in artificial intelligence for dealing with various hard combinatorial state-space search problems. Since pure best-first search techniques such as A* [3] consume a lot of time and resources for producing solutions, anytime algorithms are developed. Anytime heuristic search algorithms aim at producing good quality solutions quickly and improve upon them as time passes, with the help of heuristic guidance.

Depth-first branch and bound [5] with heuristic guidance can be viewed as a simple anytime heuristic search algorithm. More involved anytime algorithms include: Beam-stack search [10] which makes beam search complete via chronological backtracking and is sometimes referred to as depth-first beam search; Anytime Window A* (AWA*) [1] which uses a window-based depth-guiding mechanism over best-first search to produce solutions in an anytime manner; and Anytime Non-parametric A* (ANA*) [9] which improves upon the parametric weighted-A* based anytime techniques to give rise to a greedy non-parametric version.

We develop a new anytime algorithm inspired by the working of beam search [7] and structural anytime algorithms such as AWA*. The idea is to proceed in a best-first manner over a limited set of nodes– similar to beam search to produce solutions quickly, while ensuring completeness by avoiding inadmissible pruning– similar to structural anytime algorithms. This is achieved by focusing on a selected set of seed nodes in each iteration of the algorithm which we call as *Pack*. The size of the pack can be either given as input or varied using standard mechanisms both of which are demonstrated. Also, the first iteration of the proposed algorithm matches with that of beam search but without the inadmissible pruning. Experimental results indicate that the proposed algorithm gives very good anytime performance compared to the above discussed algorithms.

The rest of the paper is organized as follows: In Section 2, we present the proposed algorithms and their properties. In Section 3, we present the experimental results comparing the proposed algorithms with some of the state-of-the-art anytime heuristic search algorithms, and conclude with the key observations.

2 Proposed Methods

In this section, we present the proposed anytime pack search algorithm and its variations, along with their properties.

Anytime Pack Search (APS) takes as input the search graph and an initial state or start node, similar to other anytime algorithms. Additionally, it takes the *pack size* K as input which is a parameter specific to APS. This parameter can be controlled by the user and it determines the frequency at which solutions are produced by the algorithm. Using a small value of K, one can expect solutions to be reported at a high frequency, and vice-versa.

Algorithm 1. Anytime Pack Search (APS)

```
1: INPUT :: A search graph $G$, a start node $s$, and pack size $K$.
2: $BestSol \leftarrow \infty; g(s) \leftarrow 0;$ Calculate $f(s); Level(s) \leftarrow 0;$
   $SuspendList \leftarrow \{s\}; ClosedList \leftarrow \phi; OpenList \leftarrow \phi; ChildrenList \leftarrow \phi;$
3: while $SuspendList \neq \phi$ do
4:   $\{BestSol, Path\} \leftarrow$ ExpandKNodes($SuspendList, K, BestSol, other\ lists$);
5:   while $ChildrenList \neq \phi$ do
6:     Move all nodes from $ChildrenList$ to $OpenList$; (or, equivalently, swap the lists)
       $\{BestSol, Path\} \leftarrow$ ExpandKNodes($OpenList, K, BestSol, other\ lists$);
7: return $BestSol, Path$;
```

Algorithm 1 presents the pseudo-code of APS. It uses four lists for its operation: *SuspendList*, *ClosedList*, *OpenList*, and *ChildrenList*. Initially all lists are empty except for *SuspendList* which contains the start node. In each iteration (Lines 4 to 6 of Algorithm 1), initially, K nodes are expanded (the process of generating children) from *SuspendList* which act as a set of seeds for finding a solution in that iteration. The generated children go to *ChildrenList* whose size is capped at K, the remaining of which go to *SuspendList*. Consequently, *OpenList* takes the K nodes from *ChildrenList* for expansion, whose children will go to *ChildrenList* similar to what has been mentioned above. This process will continue until *ChildrenList* is empty which *will* happen since the search is depth-bounded like that of most anytime algorithms. APS terminates when there are no more nodes remaining for expansion. Each iteration can be seen as an attempt towards producing a better solution by exploring a beam originating from a most promising subset of the unexpanded nodes.

Algorithm 2. ExpandKNodes

```
1: INPUT :: $CurList$ from which $K$ nodes are to be expanded, $K$, $BestSol$, and the other lists.
2: for $K$ number of times do
3:   if $CurList = \phi$ then
4:     return $\{BestSol, Path\}$;
5:   $n \leftarrow$ least f-valued node from $CurList$; (for minimization problem)
6:   if IsGoal($n$) then
7:     if $f(n) < BestSol$ then
8:       $BestSol \leftarrow f(n); Path \leftarrow$ Path from $s$ to $n$ obtained by tracing the parent of $n$;
9:     Move $n$ from $CurList$ to $ClosedList$; continue;
10:  GenerateChildren($n$); Move $n$ from $CurList$ to $ClosedList$;
11: return $\{BestSol, Path\}$;
```

Algorithm 2 presents the pseudo-code of the ExpandKNodes routine. Most promising node is chosen for expansion each time from the given list which demonstrates the localized best-first nature of APS. If the chosen node is a goal node, the current best solution is updated, otherwise, its children are generated and the node is added to the $ClosedList$.

Algorithm 3. GenerateChildren

```
1: INPUT :: Node $n$ whose children are to be generated, $K$, and the lists.
2: if $Level(n) = MAX\_DEPTH - 1$ then
3:   return;
4: for each successor $n'$ of $n$ do
5:   if $n'$ is not in any of the lists then
6:     $Level(n') \leftarrow Level(n) + 1$; Insert $n'$ to $ChildrenList$;
7:   else if $g(n') <$ its previous $g$-value then
8:     Update $Level(n'), Parent(n'), g(n'), f(n')$; Move $n'$ to $ChildrenList$;
9:   if $|ChildrenList| > K$ then
10:    Move a node with largest f-value from $ChildrenList$ to $SuspendList$;
```

Finally, Algorithm 3 presents the pseudo-code of the GenerateChildren routine. It generates the children within the given depth bound MAX_DEPTH. Each child is checked as to whether it is already present in the memory, updated with the shortest path from start node, and it is inserted/moved to $ChildrenList$, similar to what is done by all the search algorithms. Whenever the size of $ChildrenList$ exceeds K, a least promising node is moved to $SuspendList$, which is a feature specific to APS.

Additionally, when using admissible heuristics, one can prune the nodes whose f-value exceeds that of the best known solution. Next, we present the properties satisfied by the proposed method.

Property 1: APS is complete and guarantees termination with an optimal solution, provided MAX_DEPTH is at-least as large as the number of nodes on a minimum-length optimal solution path.

Property 2: APS expands at-most $K \times MAX\_DEPTH$ number of nodes in each iteration, where K is the pack size.

The above property holds since the depth of shallowest nodes in *OpenList* increases by 1 in each step of the iteration (to a maximum of $MAX\_DEPTH - 1$), and each step involves expansion of at-most K nodes.

An interesting variation of the proposed algorithm is **Anytime Pack Progressive Search (APPS)**. In this variation, we initialize the value of *pack size* K with $INIT$ and increase it in each iteration by $STEP$ while it is less than $BOUND$. Typical values of $INIT$, $STEP$, and $BOUND$ are 1, 1, and 100 (or some constant) respectively. $BOUND$ helps in converging to the solutions in a time-bounded manner.

Another useful variation is named **Anytime Pack Scaling Search (APSS)**. This variation proceeds in a similar manner to APPS, except that whenever a better solution is found by the algorithm, the *pack size* K is reset to $INIT$, with the hope that the next better solution may be found with only a minimal effort (number of node expansions).

3 Experimental Results

Now, we present the results comparing APS and its variations against Depth-first Branch and Bound (DFBB) [5], Beam-Stack search (BS) [10], Anytime Window A* (AWA*) [1], and Anytime Non-parametric A* (ANA*) [9]. Since the performance of Beam-Stack algorithm varies with the beam-width, we have tested it with multiple values and presented the best results obtained amongst all of them. All the experiments have been performed on a Dell Precision T7600 Tower Workstation with Intel Xeon CPU E5-2687W at 3.1-GHz $\times$ 16 and 256-GB RAM. We display the results in terms of a metric called *% Optimal Closeness* (measures closeness to the optimal solution for minimization problems; whenever optimal solution is known), which is defined as: $\%\ Optimal\ Closeness = (Optimal\ \ solution/Obtained\ \ solution) \times 100$.

Sliding-tile Puzzle Problem (SPP): For our experiments, we have considered all 50 24-puzzle instances from [4, Table II]. Manhattan distance heuristic

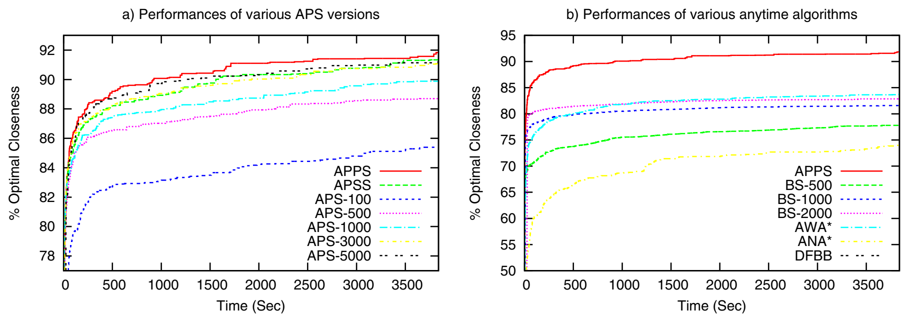

Fig. 1. Comparison of the average of anytime performances on the 50 Puzzle instances. DFBB could not come up with any solution in most cases in the given time.

is used as the heuristic estimation function (which underestimates the actual distance to goal). All algorithms are allowed to explore up-to a maximum depth of 1000 (around 10 times the depth of optimal solutions).

Figure 1a shows the comparison of anytime performances of different APS versions on the Puzzle instances. Here APPS and APSS are run with $INIT$, $STEP$, and $BOUND$ being 1000, 1000, and 5000 respectively. APS-x denotes that APS is run with pack size x. Figure 1b shows the comparison of anytime performance of best performing APS version– APPS, with that of the other anytime algorithms where it can be seen that APPS clearly stands out better. Here, BS-x denotes that beam-stack search is run with beam-width x.

Traveling Salesman Problem (TSP): We chose the first 50 symmetric TSPs (when sorted in increasing order of their sizes) from the traveling salesman problem library (TSPLIB) [8] for our experiments. These range from burma14 to gr202 where the numerical postfixes denote the size of the TSPs. Minimum spanning tree (MST) heuristic is used as the heuristic estimation function (which is an under-estimating heuristic).

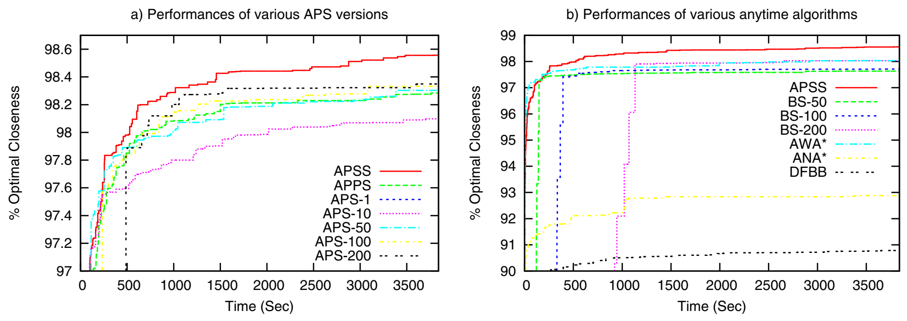

Fig. 2. Comparison of the average of anytime performances on the 50 TSP benchmarks

Figure 2a shows the comparison of anytime performances of different APS versions on the TSP instances. Here APPS and APSS are run with $INIT$, $STEP$, and $BOUND$ being 1, 1, and ∞ (no bound) respectively. Figure 2b shows the comparison of anytime performance of best performing APS version– APSS, with that of the other anytime algorithms where once again it can be seen that the APS version outperforms the others.

Single-Machine Scheduling Problem (SMS): We have generated 48 SMS problem instances[1] with different tardy factors and range factors using similar settings as given in [6]. Each instance contains 100 jobs. For each job j, an integer processing time p_j, earliness penalty h_j, and tardiness penalty w_j are generated from the uniform distribution $[1, 10]$. Let P be the sum of processing times of all the jobs. For each job j, an integer due date is generated from the uniform distribution $[P(1 - T - R/2), P(1 - T + R/2)]$, where T is

[1] The SMS benchmarks used in this paper are available at `https://sites.google.com/site/satyagautamv/resources/sms-benchmarks`

the tardy factor, set at $0.0, 0.2, 0.4, 0.6, 0.8$ and 1.0, and R is the range factor, set at $0.2, 0.4, 0.6$ and 0.8. For each combination of parameters, two problems are generated. The lower bound computation method proposed in [6] is used for computing the under-estimating heuristic over the unscheduled jobs. Here, since the optimal solutions are not available, we display the results in terms of a metric called $\%$ *Comparative Closeness* (measures closeness to the best known solution), which is defined as: $\%\ Comparative\ Closeness = (Best\ Known\ solution/Obtained\ solution) \times 100$. We obtained the best known solution for each instance by finding the best result produced amongst all the algorithms tested by us for that instance.

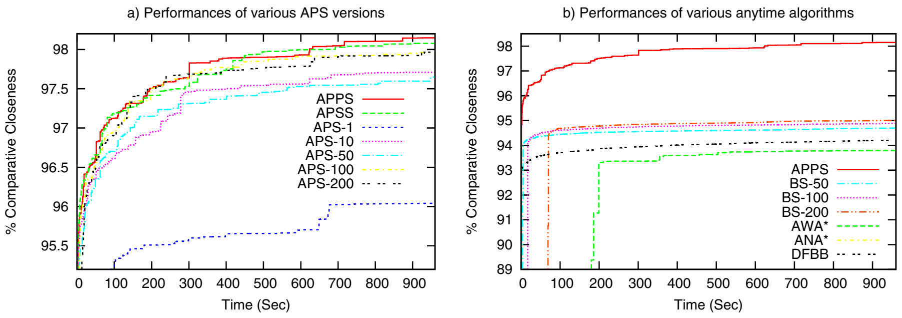

Fig. 3. Comparison of the average of anytime performances on the 48 SMS problem instances. ANA* could not come up with any solution in most cases in the given time.

Figure 3a shows the comparison of anytime performances of different APS versions on the SMS problem instances. APPS and APSS are run with $INIT$, $STEP$, and $BOUND$ being 1, 1, and ∞ (no bound) respectively. Figure 3b shows the comparison of anytime performance of best performing APS version–APPS, with that of the other anytime algorithms where the APS version dominates the others.

In conclusion, the proposed anytime heuristic search technique shows a lot of promise in solving hard combinatorial problems of both AI and Optimization domains. The choice of K (pack size) often provides a trade-off between the time taken for producing initial solution and the quality of anytime performance later on. APPS and APSS take advantage of faster initial solutions (obtained using lower K values) and better later solutions (obtained using higher K values) while avoiding the individual defects of smaller or larger K values and also making the algorithm independent of the choice of K. Experimental results over the three domains considered show that the progressive/scaling algorithms perform better than the other versions (using a specific K) and the existing algorithms.

References

1. Aine, S., Chakrabarti, P.P., Kumar, R.: AWA* - A window constrained anytime heuristic search algorithm. In: Veloso, M.M. (ed.) IJCAI, pp. 2250–2255 (2007)
2. Chakrabarti, P.P., Aine, S.: New approaches to design and control of time limited search algorithms. In: Chaudhury, S., Mitra, S., Murthy, C.A., Sastry, P.S., Pal, S.K. (eds.) PReMI 2009. LNCS, vol. 5909, pp. 1–6. Springer, Heidelberg (2009)
3. Hart, P.E., Nilsson, N.J., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. IEEE Transactions on Systems Science and Cybernetics 4(2), 100–107 (1968)
4. Korf, R.E., Felner, A.: Disjoint pattern database heuristics. Artif. Intell. 134(1-2), 9–22 (2002)
5. Lawler, E.L., Wood, D.E.: Branch-and-bound methods: A survey. Operational Research 14(4), 699–719 (1966)
6. Li, G.: Single machine earliness and tardiness scheduling. European Journal of Operational Research 96(3), 546–558 (1997)
7. Lowerre, B.: The Harpy Speech Recognition System. PhD thesis, Carnegie Mellon University (1976)
8. Reinelt, G.: TSPLIB - A traveling salesman problem library. ORSA Journal on Computing 3, 376–384 (1991)
9. van den Berg, J., Shah, R., Huang, A., Goldberg, K.Y.: Anytime nonparametric A*. In: AAAI, pp. 105–111 (2011)
10. Zhou, R., Hansen, E.A.: Beam-stack search: Integrating backtracking with beam search. In: Proceedings of the 15th International Conference on Automated Planning and Scheduling (ICAPS 2005), Monterey, CA, pp. 90–98 (2005)