Potential Search: A Bounded-Cost Search Algorithm

Roni Stern
Information Systems Engineering
Ben Gurion University
Beer-Sheva, Israel
roni.stern@gmail.com

Rami Puzis
Deutsche Telekom Laboratories
Information Systems Engineering
Ben Gurion University
Beer-Sheva, Israel
puzis@bgu.ac.il

Ariel Felner
Information Systems Engineering
Deutsche Telekom Laboratories
Ben Gurion University
Beer-Sheva, Israel
felner@bgu.ac.il

Abstract

In this paper we address the following search task: find a goal with cost *smaller than or equal to* a given fixed constant. This task is relevant in scenarios where a fixed budget is available to execute a plan and we would like to find such a plan while minimizing the search effort. We introduce an algorithm called *Potential search* (PTS) which is specifically designed to solve this problem. PTS is a best-first search that expands nodes according to the probability that they will be part of a plan whose cost is *less than or equal to* the given budget. We show that it is possible to implement PTS even without explicitly calculating these probabilities, when a heuristic function and knowledge about the error of this heuristic function are given. In addition, we also show that PTS can be modified to an anytime search algorithm. Experimental results show that PTS outperforms other relevant algorithms in most cases, and is more robust.

Introduction and Overview

Most heuristic search algorithms measure the quality of their solution by comparing it to the optimal solution. They can be classified into four major classes according to the quality of the solution that they return.

(1) Optimal algorithms. Optimal algorithms return a solution that is guaranteed to be optimal. Algorithms from this type are usually variants of the well-known A* (Pearl 1984) or IDA* (Korf 1985) algorithms. In many real-life problems it is not practical to use optimal algorithms, as many problems are very hard to solve optimally.

(2) Suboptimal algorithms. Suboptimal algorithms guarantee that the solution returned is no more than w times the optimal solution, where $w > 1$ is a predefined parameter. These algorithms are also called w-admissible. Weighted A* (Pohl 1970) and Optimistic Search (Thayer and Ruml 2008) are examples of algorithms of this class. Suboptimal algorithms usually run faster than optimal algorithms, trading the quality of the solution for running time.

(3) Any solution algorithms. Any solution algorithms return a solution, but they have no guarantee about the quality of the solutions they find. Such algorithms usually find a solutions faster than algorithms of the first two classes, but possibly with lower quality. Examples of any solution algorithms include Depth-first-branch-and-bound (DFBnB) (Zhang and Korf 1995), beam search variants (Furcy and Koenig 2005), Hill climbing and Simulated annealing.

(4) Anytime algorithms. Anytime algorithms are: "algorithms whose quality of results improves gradually as computation time increases" (Zilberstein 1996). An anytime search algorithm starts as an any solution algorithm. After the first solution is found, an anytime search algorithm continue to run, finding solutions of better qualities (with or without guarantee on their suboptimality). Some anytime algorithms are guaranteed to converge to finding the optimal solution if enough time is given. Prominent examples of anytime search algorithms are Anytime Weighted A* (Hansen and Zhou 2007) and Anytime Repairing A* (Likhachev, Gordon, and Thrun 2003).

Bounded-Cost Search Problems

In this paper we deal with a fifth type of search algorithm, addressing the following scenario. Assume that a user has a given constant amount of budget C to execute a plan. The cost of the optimal solution or the amount of suboptimality is of no interest and not relevant. Instead, a plan with cost *less than or equal to* C is needed as fast as possible. We call this problem the *bounded-cost search* problem, and define it formally as follows:

Definition 1 Bounded-cost search problem. *Given a description of a state space, a start state s, a goal test function and a constant C, a bounded-cost search problem is the problem of finding a path from s to a goal state with cost less than or equal to C.*

For example, consider an application server for an online travel agency such as Expedia, and a customer that requests a flight to a specific destination arriving before a given time and date (in which the customer has a meeting). This is clearly a bounded-cost problem, where the cost is the arrival time. The task of Expedia is to build as fast as possible an itinerary in which the user will arrive on time. The user is not concerned with the optimality or suboptimality of the resulting plan, and Expedia would like to respond quickly with a fitting solution.

None of the algorithms from the classes above directly answer this problem. Of course, ideally, one might run an

optimal algorithm. If the optimal solution cost is *less than or equal to* C then return it, otherwise return *failure*, as no solution of cost C exists. One could even use C for pruning purposes, and prune any node n with $f(n) \geq C$. However, this technique for solving the bounded-cost search problem might be very inefficient as finding a solution with cost C can be much more easy than finding the optimal solution. Similarly, it is not clear how to tune any of the suboptimal algorithms (for example, which weight to use in Weighted A* and its variants), as the cost of optimal solution is not known and therefore the ratio between the cost of the desired solution C and the optimal cost is unknown too. A possible direction for solving a bounded-cost search problem is to run an anytime search algorithm and halt it when a good enough solution is found. However, solutions with costs *higher than* C may be found first even though they are of no use. The main problem with all these variants is that the desired goal cost is not used to guide the search, i.e., C is not considered when choosing which node to expand next.

It is possible to view a bounded-cost search problem as a CSP, where the desired cost bound is simply a constraint on the solution cost. However, for many problems there are powerful domain-specific heuristics, and it is not clear if general CSP solvers can use such heuristics. The potential-based approach described next is reminiscent of CSP solvers based on solution counting and solution density, where assignments that are estimated to allow the maximal number of solutions are preferred (Zanarini and Pesant 2009).

The Potential Search Algorithm

In this paper we introduce an algorithm called *Potential search* (PTS), which is specifically designed to solve a bounded-cost search problem. PTS is designed to focus on a solution that is *less than or equal* to C, and the first solution it provides meets this requirement. PTS is a best-first search algorithm that expands nodes according to the *probability* that they will be part of a plan of cost *less than or equal to* the given budget C. We denote this probability as the *potential* of a node. Of course, the exact potential of a node is unknown. Instead, we show how any given heuristic function can be used to simulate the exact potential. This is possible as long as we have a theoretical model of the relation between the heuristic and the cost of the optimal plan. Several such models are analyzed, and a general method for implementing PTS given such a model is proposed. We prove that with this method, nodes are expanded in a best-first order according to their potential. Furthermore, PTS can also be modified to run as an anytime search algorithm. This is done by iteratively running PTS with decreasing bounded costs.

Experimental results on the standard 15-puzzle as well as on the Key Player Problem in Communication (KPP-COM) demonstrate the effectiveness of our approach. PTS is competitive with the state-of-the-art anytime and suboptimal heuristic search algorithms. It outperforms these algorithms in most cases and is more robust.

Potential search

We now turn to describe the PTS algorithm in detail. Consider the graph presented in Figure 1. Assume that we are

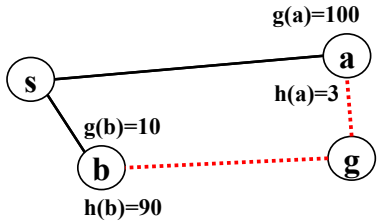

Figure 1: Example of an expansion dilemma.

searching for a path from node s to node g and that we are asked to find a path of cost *less than or equal to* 120 ($C = 120$). After expanding s, the search algorithm needs to decide which node to expand next, node a or node b.[1]

If the task were to find the optimal path from s to g, then clearly node b should be expanded first, since there may be a path from s to g that passes through b which is shorter than the cost of the path that passes through a as ($g(b) + h(b) = 100 < g(a) + h(a) = 103$). However, since any path that is shorter than 120 is acceptable in our case, expanding node b is not necessarily the best option. For example, it might be better to expand node a which is probably very close to a goal of cost less than 120 (as $h(a = 3)$).

We propose the *Potential search* algorithm (denoted as PTS) which is specifically designed to find solutions with costs *less than or equal to* C. We define the *potential* of a node as the probability (Pr) that this node is part of a path to a goal with cost *less than or equal to* C. This potential is formally defined as follows. Let $g(n)$ be the cost of the shortest path found so far from the initial state to n, and let $h^*(n)$ be the real cost of the shortest path from n to a goal.

Definition 2 : Potential. *The* potential *of node* n*, denoted* $PT(n)$*, is* $Pr(g(n) + h^*(n)) \leq C$.

PTS is simply a best-first search (or any of its variants) which orders the nodes in the open-list (denoted hereafter as OPEN) according to their potential PT and chooses to expand the node with the highest $PT(n)$.

If $h^*(n)$ is known then the $PT(N)$ is easy to calculate. It is a binary function, returning 1 if $g(n) + h^*(n) \leq C$ and 0 otherwise. Of course, usually, $h^*(n)$ is not known in advance and the exact potential of a node cannot be calculated. However, we show that it is possible to order the nodes according to their potential even without knowing or calculating the exact potential. This can be done by using the heuristic function h coupled with a model of the distribution of its values. Next we show how we can reason about the exact potential for several such heuristic models. We then show how these can be extended to the general case.

Estimating the Potential of a Node

Many years of research in the field of heuristic search have produced powerful methods for creating sophisticated

[1]Throughout this paper we use the standard heuristic search terminology, where the shortest known path between the start node s and a node n is denoted by $g(n)$, and a heuristic estimate of the distance from a node n to a goal is denoted by $h(n)$.

heuristics, such as abstractions (Larsen et al. 2010), constraint relaxation and memory based heuristics (Felner, Korf, and Hanan 2004; Sturtevant et al. 2009) as well as heuristics for planning domains (Katz and Domshlak 2010). Next, we show how it is possible to use any given heuristic and still choose to expand the node with the highest potential even without explicitly calculating it. All that is needed is knowledge about the model of the relation between a given heuristic and the optimal cost as defined in the next section.

Heuristic Models

Let h be a given a heuristic function, estimating the cost of reaching a goal from a node. Consider the relation between h and h^*. In some domains, this relation is a known property of the available heuristic function (e.g., a precision parameter of a sensor). In other domains, it is possible to evaluate the model of a heuristic function, i.e., how close h is to h^*, from attributes of the domain. In order to preserve a domain-independent perspective, we focus on several general models of this h-to-h^* relation. We call this relation the *heuristic model* or h-*model* and define it as follows:

Definition 3 h-accuracy model. *The function e is the h-model of h if $h^*(n) = e(h(n))$ for every node n.*

Note that the h-*model* is not necessarily a deterministic function, since there can be nodes with the same h but different h^* values. Next, we show that it is possible to implement PTS as a best-first search for a number of common special cases of h-*models*. The *potential* function ($PT(n)$) is not known. However, for these cases, we provide a cost function that is easy to calculate and prove that this cost function orders the nodes exactly in the order of the potential function, that is, the node with the *smallest* cost is also the node with the *highest* potential. Therefore, it is possible to *simulate* the potential of a node with this cost function.

Additive h-*Model*

Consider the following h-*model*: $h^*(n) = h(n) + X$, where X is an independent identically distributed (i.i.d.) random variable. This does not imply that the distribution of X is uniform, but just that additive error of every node is taken from the same distribution (independently). We call this type of h-*model* an *additive* h-model.[2]

Lemma 1 *For any i.i.d. random variable X, if the h-*model *is $h^*(n) = h(n) + X$ and $f(n) = g(n) + h(n)$ then for any pair of nodes n_1,n_2 we have:*

$$f(n_1) \leq f(n_2) \Leftrightarrow PT(n_1) \geq PT(n_2)$$

Proof: Assume that $PT(n_1) \geq PT(n_2)$, and let h_i, g_i and h_i^* denote $h(n_i), g(n_i)$ and $h^*(n_i)$ respectively. According to the potential definition (Definition 2) then:

$$\begin{array}{rcl} Pr(g_1 + h_1^* \leq C) & \geq & Pr(g_2 + h_2^* \leq C) \\ \Leftrightarrow Pr(h_1^* \leq C - g_1) & \geq & Pr(h_2^* \leq C - g_2) \end{array}$$

According to the h-*model*, this is equivalent to:

$$\begin{array}{rcl} Pr(h_1 + X \leq C - g_1) & \geq & Pr(h_2 + X \leq C - g_2) \\ \Leftrightarrow Pr(X \leq C - g_1 - h_1) & \geq & Pr(X \leq C - g_2 - h_2) \end{array}$$

Since X is i.i.d., then this is equivalent to:

$$C - g_1 - h_1 \geq C - g_2 - h_2 \Leftrightarrow f(n_2) \geq f(n_1) \square$$

Consequently, for any problem with an *additive* h-*model*, standard A*, which expands the node with the smallest f-value, will always expand the node with the highest potential. This results is summarized in Theorem 1:

Theorem 1 *For any i.i.d. random variable X, if h^*=$h + X$ then PTS can be implemented as a best-first search using the standard cost function of A*, $f = g + h$.*

Therefore, for an additive h-*model* we can order the nodes in OPEN according to their potential, even without knowing the exact potential and regardless of the distribution of X.

Linear Relative h-*Model*

An *additive* h-*model* may not fit many real problems. Consider for example a shortest path problem in a map, using the air distance as a heuristic. If the air distance between two nodes is very large, there is a larger possibility that obstacles exist between them. More obstacles imply larger difference between the air distance and the real shortest path. We therefore propose the following more realistic h-*model*: $h^*(n) = h(n) \cdot X$ for any random i.i.d. variable X. We call this type of model the *linear relative* h-*model*[3] and present the following cost function:

$$f_{lnr}(n) = \frac{h(n)}{C - g(n)}$$

Next we prove that nodes with smaller $f_{lnr}(n)$ are more likely to find a path with cost $\leq C$. The intuition behind this is as follows. $C - g(n)$ is an upper bound on the remaining cost to the goal from node n that may result in a path with total cost *smaller than or equal to* C. $h(n)$ is a lower bound estimation of the remaining cost. Therefore, nodes with smaller $\frac{h(n)}{C-g(n)}$ are more likely to find a path within such a bound.

Lemma 2 *For any i.i.d. random variable X, if the heuristic model is $h^*(n) = h(n) \cdot X$ then for any pair of nodes n_1,n_2 we have: $f_{lnr}(n_1) \leq f_{lnr}(n_2) \Leftrightarrow PT(n_1) \geq PT(n_2)$*

Proof: Assume that $PT(n_1) \geq PT(n_2)$. According to the potential definition this means that:

$$\begin{array}{rcl} Pr(g_1 + h_1^* \leq C) & \geq & Pr(g_2 + h_2^* \leq C) \\ \Leftrightarrow Pr(h_1^* \leq C - g_1) & \geq & Pr(h_2^* \leq C - g_2) \end{array}$$

According to the h-*model*, this is equivalent to:

$$\begin{array}{rcl} Pr(X \cdot h_1 \leq C - g_1) & \geq & Pr(X \cdot h_2 \leq C - g_2) \\ \Leftrightarrow Pr(X \leq \frac{C-g_1}{h_1}) & \geq & Pr(X \leq \frac{C-g_2}{h_2}) \end{array}$$

Since X is i.i.d., this is equivalent to:

$$\frac{C-g_1}{h_1} \geq \frac{C-g_2}{h_2} \Leftrightarrow \frac{h_1}{C-g_1} \leq \frac{h_2}{C-g_2} \Leftrightarrow f_{lnr}(n_1) \leq f_{lnr}(n_2) \square$$

Consequently, for any problem with a heuristic that has a *linear relative* h-*model*, a best-first search that uses f_{lnr} as a cost function will expand nodes exactly according to their potential. This result is summarized in Theorem 2:

[2] This is reminiscent of the *bounded constant absolute error* model described by (Pearl 1984) where the difference between h and h^* was bounded by a constant (i.e., $h^*(n) \leq h(n) + K$). Here, K is the largest values for X.

[3] This model is reminiscent of the *constant relative error* (Pearl 1984), where $h^*(n) \leq K \cdot h(n)$ for some constant K.

h-*model* (e)	$e^r(h^*,h)$	P_{gen}	f_{gen}
h+X (additive)	h^*-h	C-g-h	f=g+h
$h \cdot X$ (linear relative)	$\frac{h^*}{h}$	$\frac{C-g}{h}$	$\frac{h}{C-g}$=f_{lnr}
h^X	$log_h(h^*)$	$log_h(C\text{-}g)$	-$log_h(C\text{-}g)$

Table 1: h models and their corresponding cost functions.

Theorem 2 *For any i.i.d. random variable X, if h^*=$h \cdot X$ then PTS can be implemented as a best-first search using f_{lnr} as a cost function.*

General h-*Model*

Consider the more general h-*model*, which is some function of h and a random i.i.d. variable X. We denote this function by e and write $h^* = e(h, X)$. Let e^r be the *inverse* function of e such that $e^r(h^*, h) = X$. We denote an h-*model* as *invertible* if such an inverse function e^r exists, and define a general function $P_{gen}(n) = e^r(C - g(n), h)$ which simulates the *potential* of a node as follows.

Lemma 3 *Let X be an i.i.d. random variable X and $h^* = e(h, X)$ an invertible h-*model*, where e^r is monotonic. Then, for any pair of nodes n_1,n_2, $P_{gen}(n_1) \geq P_{gen}(n_2) \Leftrightarrow PT(n_1) \geq PT(n_2)$*

Proof: Assume that $PT(n_1) \geq PT(n_2)$.
According to the potential definition this means that:

$$Pr(g_1 + h_1^* \leq C) \geq Pr(g_2 + h_2^* \leq C)$$
$$\Leftrightarrow Pr(h_1^* \leq C - g_1) \geq Pr(h_2^* \leq C - g_2)$$

Since e is invertible, we apply $e^r(\cdot, h)$ to both sides:

$$Pr(X \leq e^r(C - g_1, h_1)) \geq Pr(X \leq e^r(C - g_2, h_2))$$

Since X is i.i.d., this is equivalent to:

$$e^r(C - g_1, h_1) \geq e^r(C - g_2, h_1)$$
$$\Leftrightarrow P_{gen}(n_1) \geq P_{gen}(n_2) \square$$

Thus, for any problem with an invertible h-*model*, a best-first search that expands the node with the *highest* P_{gen} will exactly simulate PTS as this is the node with the *highest potential*. P_{gen} can be easily converted (e.g., by multiplying P_{gen} by -1) to a cost function f_{gen} where an equivalent best-first search will choose to expand the node with the *smallest* f_{gen}. This is summarized in Theorem 3:

Theorem 3 *For any i.i.d. random variable X, if $h^* = e(h, X)$, e is invertible, and e^r is monotonic, then PTS can be implemented as a best-first search using a cost function $f_{gen}(n)$.*

Notice that Theorems 1 and 2 are in fact special cases of Theorem 3. Table 1 presents a number of examples of how Theorem 3 can be used to obtain cost functions for various h-*models*. These cost functions can then be used in a best-first search to implement PTS.

The exact h-*model* is domain dependent. Analyzing a heuristic in a given domain and identifying its h-*model* may be done analytically in some domains with explicit knowledge of the domain attributes. Another option for identifying a h-*model* is by adding a preprocessing stage in which a set of problem instances are solved optimally, and the h-*model* is discovered using curve fitting methods.

Experimental Results: 15-Puzzle

To show the applicability and robustness of PTS we present experimental results on two domains: the 15-puzzle and the Key Player Problem in Communication (KPP-COM).

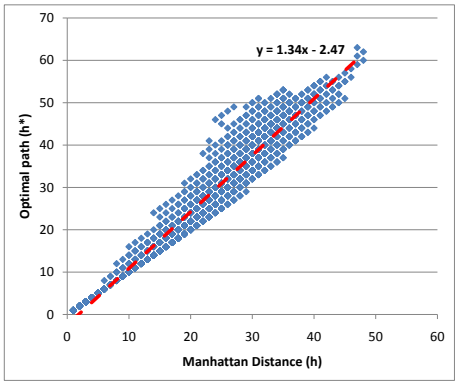

Figure 2: Manhattan distance heuristic Vs. true distance.

There are many advanced heuristics for the 15-puzzle, but we chose the simple Manhattan distance heuristic (MD) as our goal is to compare search algorithms and not different heuristics. In order to choose the correct h-*model* for MD, we sampled 50 of the standard 100 random instances (Korf 1985) and solved each instance optimally. For every such instance we considered all the states on the optimal path, to a total of 2,507 states. Each of these states were assigned a 2-dimensional point, where the x value denotes the MD of the state and the y value denotes its optimal cost to the goal. The plot of these points is presented in Figure 2. The dashed red line indicates the best linear fit for the plot, which is the line $y = 1.34 \times x - 2.47$. It is easy to observe that linear fit is very close and therefore we solved the 15-puzzle with *Potential Search* using a *linear relative* h-*model* and its corresponding implementation using f_{lnr} (Theorem 2).

PTS was implemented on the 15-puzzle to solve the bounded-cost search problem. In addition, we also implemented a number of state-of-the-art anytime algorithms but focus here on the two anytime algorithms that performed best: Anytime Weighted A* (Hansen and Zhou 2007) and Optimistic Search (Thayer and Ruml 2008), denoted as AWA* and OS respectively. AWA* and OS are anytime algorithms but they can be easily modified to solve the bounded-cost search problem by halting these algorithms when a solution with cost *less than or equal to* the desired cost C was found, and pruning nodes with $g + h > c$.

AWA* is a simple extension of Weighted A* (Pohl 1970) (WA*). WA* orders the nodes in OPEN according to the cost function $f = g + w \cdot h$, were w is a predefined parameter. After the first solution has been found AWA* simply continues to run WA*, finding goals with better costs. OS is a suboptimal search algorithm which uses two cost functions: an admissible heuristic h and an inadmissible (but possibly more accurate) heuristic $\hat{h}$. In our experiments, $\hat{h}$ was implemented as a weighted version of h, i.e. $\hat{h} = w \cdot h$ where w is a predefined parameter. OS chooses to expand the node with the lowest $g + \hat{h}$ but switches to using $g + h$

C	OS-1.50	OS-2.00	OS-3.00	PTS	AWA*-1.50	AWA*-2.00	AWA*-3.00
55	**13%**	28%	63%	23%	14%	29%	68%
60	**11%**	21%	74%	12%	**11%**	25%	60%
65	6%	17%	40%	**4%**	6%	14%	41%
70	6%	**3%**	9%	**3%**	6%	4%	9%
75	6%	3%	3%	**2%**	6%	3%	4%
80	6%	3%	3%	**2%**	6%	3%	**2%**
85	6%	3%	2%	**1%**	6%	3%	2%
90	6%	3%	2%	**1%**	6%	3%	**1%**

Table 2: Expanded nodes as a percentage of nodes expanded by A*. Fixed desired cost C.

if all the nodes in OPEN will not improve the *incumbent solution* (=best solution found so far) according to $\hat{h}$. OS was shown to be highly effective for many domains (Thayer and Ruml 2008). In its basic form, OS is a suboptimal algorithm, halting the search when the ratio between the lowest $g + h$ in the openlist and the incumbent solution is below a desired suboptimality. However, it can easily be extended to an anytime algorithm by simply continuing the search process until the desired goal cost is found.

We performed two sets of experiments which differ in the way the desired cost C was calculated. In the first set of experiments we ran PTS, OS and AWA* (with different w) on 75 random 15-puzzle instances with a fixed desired cost C of 90, 85, 80 down to 55. The exact same cost C was set to *all* instances, no matter what was the optimal solution cost. Table 2 presents the average number of nodes expanded until a goal with a cost equal to or under the desired cost was found, as a percentage of the number of nodes expanded by A* when finding the optimal solution. Every row corresponds to different desired goal cost. The algorithm with the lowest number of expanded nodes in every row is marked in bold. For OS and AWA*, we experimented with w=1.5, 2 and 3. As can be seen, for desired goal costs of 55 and 60, OS-1.5 expands the fewest nodes. In all other cases PTS outperforms both algorithms. Furthermore, even for cost 55 and 60, PTS performs relatively well, expanding only 23% and 12% of nodes expanded by A*, respectively. This demonstrates the robustness of PTS.

In the second set of experiments the desired cost C was different for each individual instance and was based on the optimal solution cost as follows. First, we found the optimal solution with A* and MD for 75 random instances. Then, for every instance we ran PTS, OS and AWA* with a desired cost C that was set to be a factor of 1.1,..,1.9 times the optimal cost. All algorithms were halted when a solution of cost *less than or equal to* C was found. Both OS and AWA* have a parameter w which was set to 1.5, 2 and 3. Table 3 presents the average number of expanded nodes, for the different algorithms (using MD as a heuristic) and different bounds. Bold fonts mark the best algorithm in terms of minimum number of nodes expanded. Since C was different for each instance, the C column in the table gives the degree of suboptimality ($1 + w \times optimal$) that was used to calculate C. Runtime results are omitted since they show exactly the same trends as the number of expanded nodes. This is reasonable since all algorithms use a cost function that is a simple arithmetic operation of g and the same heuristic function (MD), and all algorithms are best-first searches, implemented using a priority queue based openlist, and thus the time per expanded node is practically the same for all the algorithms.[4]

As can be seen, for different desired cost bounds C different algorithms perform best. For constants that are close to 1 (suboptimality of 1 up to 1.3), OS-1.50 or OS-2.00 are the best. For large constants (of high suboptimality levels), either PTS or OS-3.00 performs best.

All this is meaningful if we know the cost of the optimal solution and one can therefore choose the best variant of either OS or AWA*. However, the bounded-cost problem assumes that there is no knowledge about the optimal cost, as is often the case in reality. Therefore, in such cases one would not know how to choose the weight that will result in best performance. One is thus forced to guess a value for w without knowing the cost of the optimal solution and as a consequence without knowing the degree of suboptimality. This means that each individual column should be compared as a stand alone algorithm to any other column. The table clearly shows that for both OS and AWA* any given value of w we tried (individual columns) performs best or reasonable in only a very limited range. For example, OS-2 performs well only when the desired solution is within a degree of suboptimality of 1.2 or 1.3. Guessing $w = 2$ and thus using OS-2 will only perform well in such ranges of solutions but will perform much worse in other ranges.

By contrast, PTS is much more robust. PTS is clearly superior when compared to any given fixed setting (any possible column) for both OS and AWA* across the different values of the desired cost C (rows). It is only outperformed by OS-1.50 for C that corresponds to suboptimality of 1 and 1.1, by OS-2.00 for suboptimality of 1.2 and 1.3 and by AWA*-3.00 for suboptimality of 1.6-1.8. In all other cases PTS outperforms all other variants. Furthermore, PTS was the best variant and outperformed **all** other variants for suboptimality levels 1.4, 1.5 and 1.9. In all other suboptimality levels PTS was relatively close to the best algorithms. Therefore, we can conclude from this set of experiments that if the suboptimality level is not known PTS is clearly the algorithm of choice.

Note that PTS is the only algorithm where the number of expanded nodes continues to decrease as the desired cost bound increases, speeding up the search with respect to the desired bound. By contrast, for any weight shown in Table 3, the number of nodes expanded by either AWA* or OS

[4]OS is a slight exception, since it maintains two openlists. However, we have seen that the overall runtime trends remain the same for OS as well.

C	OS-Oracle	AWA*-Oracle	PTS	OS-1.50	OS-2.00	OS-3.00	AWA*-1.50	AWA*-2.00	AWA*-3.00
1	4,538,762	2,555,737	2,048,601	**881,563**	1,408,569	2,162,069	1,437,363	2,554,212	3,857,002
1.1	1,792,151	1,648,738	258,883	**114,619**	283,863	1,125,460	124,218	553,348	1,261,070
1.2	833,784	819,627	78,949	108,557	**45,760**	631,379	124,562	66,514	696,098
1.3	458,601	396,407	42,607	108,557	**28,372**	275,422	124,619	30,949	196,723
1.4	198,933	218,323	**25,764**	108,557	26,226	38,259	124,619	26,859	69,824
1.5	108,557	124,619	**18,394**	108,557	26,227	20,199	124,619	26,948	24,060
1.6	76,211	86,035	17,069	108,557	26,227	13,777	124,619	26,948	**11,803**
1.7	62,198	61,545	12,763	108,557	26,227	**10,970**	124,619	26,948	12,107
1.8	48,243	46,965	11,896	108,557	26,227	**10,833**	124,619	26,948	10,998
1.9	43,345	40,782	**10,559**	108,557	26,227	10,834	124,619	26,948	10,996

Table 3: 15-puzzle expanded nodes. The desired cost C was based on a suboptimality degree.

decreases with the cost bound only until a certain point, after which the number of expanded nodes remains constant (excluding the "oracle" variants which will be discussed in the next paragraph).[5] The explanation is as follows. Let AWA*(w) be AWA* with weight w, and let C_w be the cost of the first goal found by AWA*(w). Clearly, AWA*(s) will expand the same set of nodes until the first goal is found, regardless of the desired bound C. Thus the number of nodes expanded by AWA*(w) will be exactly the same, for any desired cost $C \geq C_w$. Similar argument applies for OS.

Assume an oracle that provides the optimal solution cost. In this case, one could calculate the exact ratio between C and the optimal cost and then set this ratio as input for a suboptimal algorithm. The OS-Oracle and AWA*-Oracle columns in Table 3 present the results where we set w for AWA* and OS to be exactly the desired suboptimality, e.g., for suboptimality of 1.5 we set $w = 1.5$. PTS clearly outperforms this "oracle" variants. The reason can be explained by the known phenomenon for WA* variants, where for a given weight w, the quality of the returned solution is much better than just w times optimal (Hansen and Zhou 2007). Therefore, if one wants to find a solution of suboptimality of w then a parameter larger than w should be used in order to get the best performance. For example, if one wants a solution with guaranteed suboptimality of 1.5 then OS with $w = 3$ (displayed in the column OS-3.00) is a better choice (20,199 nodes) than OS with $w = 1.5$ (108,557 nodes).

Experimental Results: Key Player Problem

Our first domain (the 15-puzzle) represents the class of domains where one wants to find a path (or solution) of *minimal* cost. In order to test the algorithms on a diversity of domains, for our second domain we chose a problem where the search is aimed at finding a solution with the *maximal* utility. We thus implemented the three algorithms on the *Key Player Problem in Communications* (KPP-COM) (Puzis, Elovici, and Dolev 2007) which was shown to be NP-Complete.

KPP-COM is the problem of finding a set of k nodes in a graph with the highest *group betweenness centrality* (GBC). (GBC) is a metric for centrality of a group of nodes (Everett and Borgatti 1999). It is a generalization of the *betweenness* metric that measures the centrality of a node with respect to the number of shortest paths that pass through

[5]For OS-3.0 and AWA*-3.0 the number of expanded nodes for $C = 1.8$ and $C = 1.9$ are practically the same.

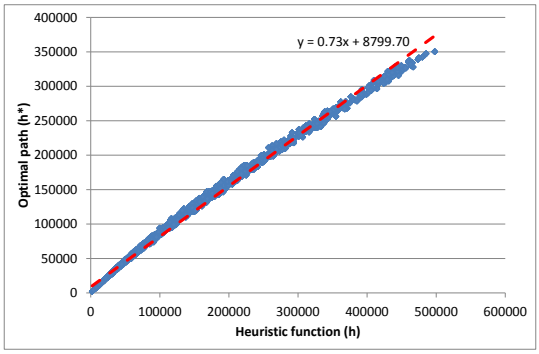

Figure 3: KPP-COM optimal solution vs. heuristic.

it (Freeman 1977). Formally, the betweenness of a node n is $C_b(n) = \sum_{s,t \in V s,t \neq n} \frac{\sigma_{st}(n)}{\sigma_{st}}$, where σ_{st} is the number of shortest paths between s and t and $\sigma_{st}(n)$ is the number of shortest paths between s and t that passes through n. The betweenness of a group of nodes A, termed group betweenness, is defined as $C_b(A) = \sum_{s,t \in V \setminus A} \frac{\sigma_{st}(A)}{\sigma_{st}}$ where $\sigma_{st}(A)$ is the number of shortest paths between s and t that pass through at least one of the nodes in A.

KPP-COM can be solved as a search problem. Let $G = (V, E)$ be the input graph in which we are searching for a group of k nodes with highest GBC. A state in the search space consists of a set of vertices $N \subseteq V$ which are considered to be the group of vertices with the highest GBC. The initial state of the search is an empty set, and a children of a state correspond to adding a single vertex to the set of vertices of the parent state. Instead of a cost, every state has a utility, which is the GBC of the set of vertices that it contains. Note that since in this problem the optimal solution has the *maximal* GBC, an admissible heuristic is required to be an *upper bound* on the optimal utility. Similarly, a suboptimal solution is one with smaller utility than the optimal.

A number of efficient admissible heuristics for this problem exist (Puzis, Elovici, and Dolev 2007) and in our experiments we used the best one, which is calculated as follows. Consider a node that consists of a set of m vertices V_m. First, the contribution of every individual vertex $v \in V$ that is not in V_m is calculated. This is the GBC of $V_m \cup \{v\}$ minus the GBC of V_m. Then, the contribution of the topmost

C	PTS	DFBnB	OS-0.7	OS-0.8	OS-0.9	AWA*-0.7	AWA*-0.8	AWA*-0.9	AWA*-1.0
280,000	**329**	347	338	439	803	336	438	796	1,750
290,000	**408**	422	429	584	1,167	424	580	1,155	2,729
300,000	**556**	562	575	770	1,547	569	758	1,528	4,118
310,000	**727**	735	749	989	1,876	751	972	1,850	4,389
320,000	1,194	**1,117**	1,145	1,396	2,746	1,138	1,391	2,713	7,229

Table 4: Average runtime in seconds on KPP-COM instances.

$k - m$ vertices is summed and used as an admissible heuristic. We denote this heuristic as h_{GBC} (see (Puzis, Elovici, and Dolev 2007) for more detailed discussion on this heuristic). Since the main motivation for the KPP-COM problem is in communication network domains, all our experiments were performed on graphs generated by the model provided by (Barabasi and Albert 1999). This model is a well-used model of Internet topology and the web graph. First, we searched for a fitting *h-model* for the h_{GBC} heuristic. Figure 3 shows the h^* (which is the maximum utility that can be added to a node) as a function of the h_{GBC} heuristic. This was calculated by solving 100 random instances optimally, and backtracking from the solution node to the root node. The real distance to the goal (in terms of utility) of the nodes on the optimal paths as a function of their heuristic values is plotted in Figure 3. The dashed red line is a linear fit of the data. As can be seen, this heuristic also exhibits a clear *linear relative h-model*. Thus, we used the f_{lnr} cost function for implementing PTS.

We performed the following experiments on this problem. First, a graph with 600 nodes was generated according to the Barabási-Albert model, with a density factor of 2. Then we ran PTS, AWA* and OS (both with different weights) given a desired costs of 250,000, 260,000, ... , 320,000, limiting the size of the searched group of vertices to be 20 (i.e., $k = 20$). The average optimal utility was 326,995. Since the depth of the solution is known in advance (the size of the searched group k), we also ran *Depth-first branch and bound* (DFBnB), which is known to be highly effective when the depth of the solution is known and many solutions exist. This experiment was repeated 25 times and Table 4 presents the average runtime in seconds until a node with utility *larger than or equal to* the desired utility was found.

Indeed, PTS is shown to be effective for all of the desired utilities, performing slightly better than DFBnB, which is known to perform very well on this domain (Puzis, Elovici, and Dolev 2007). Notice that in this problem, low weights, used by AWA* will also achieve very good performance and converge to DFBnB. This is because a low weight to the heuristic causes the search to focus more the g part of the cost function $f = g + w \cdot h$, resulting in a DFBnB-like behavior where deeper nodes are preferred.

Potential Search as an Anytime Algorithm

PTS can be modified to be an anytime search algorithm which we call *anytime potential search* (APTS). APTS uses the following greedy approach to anytime search: focus on finding a solution that is better than the incumbent solution (=best solution found). This can be naturally implemented using PTS. Simply set C to be the cost of the incumbent solution, minus a small constant ϵ, which can be the smallest edge cost in the graph. This is iteratively repeated until APTS fails to find better solutions, in which case the optimal path to a goal has been found. It is even possible to transfer the OPEN and CLOSED lists of APTS between iterations, recalculating the potential cost function for all the nodes in OPEN, when a new goal node with better cost is found. Of course, this has time overhead as all the nodes must be reinserted to OPEN with their new cost (e.g., f_{lnr}).

Experimental Results

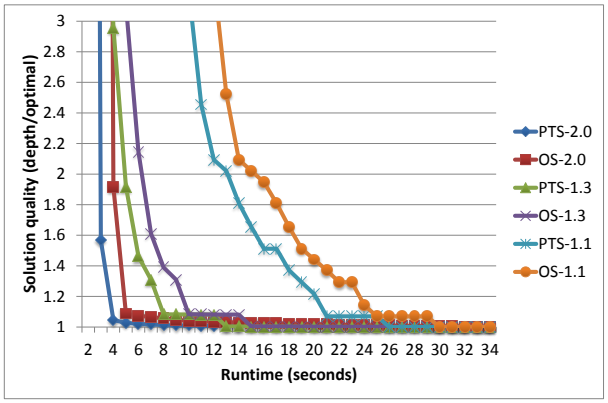

Figure 4: 15-puzzle, solution quality Vs. Runtime.

We experimented on all the standard 100 random 15-puzzle instances. Figure 4 shows the results of APTS Vs. OS. The x-axis denotes the runtime and the y-axis displays the solution quality (the depth of goal found divided by the optimal goal depth). The rightmost point in the x-axis denotes the average runtime required for A* to find the optimal solution. Again, we used $f = g + w \cdot h$ as an inadmissible cost function for OS. As explained above, APTS performs a series of PTS iterations, each with a different desired cost C. To initiate C for the first iteration, we first ran WA* with the same parameter w used for OS until the first solution was found. Then, APTS was activated. As can be seen in Figure 4, when using the same weight APTS always outperforms OS. We also compared APTS to AWA* with various weights, and found that AWA* and APTS have very similar performance in this domain.

We also performed experiments with APTS on the KPP-COM problem. Since any subset of vertices has a GBC, then every node generated induces a (probably suboptimal) utility. Thus even before A^* expands a goal (and returning the optimal solution), it can return suboptimal solutions. Therefore in KPP-COM APTS does not require any parameter in order to find an initial goal fast (as opposed to the 15-puzzle,

in which an initial WA* was performed).

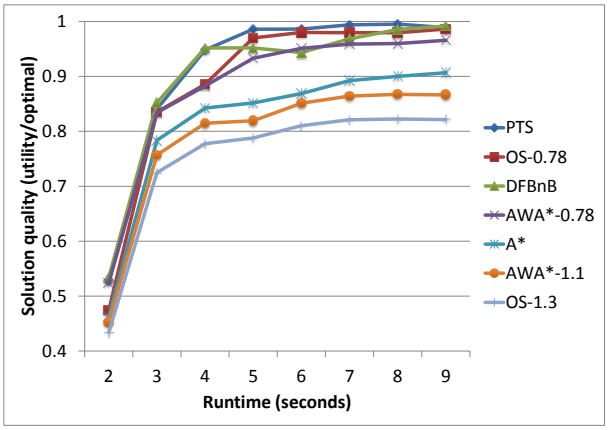

Figure 5: Anytime KPP-COM, 600 nodes, density 2.

Figure 5 displays average results on 100 different graphs with 600 nodes, density of 2 and a desired group size of 20. The x-axis denotes the runtime and the y-axis the solution quality (best utility found divided by the optimal utility). As can be seen in Figure 5 APTS (without any parameter) outperforms any other algorithm. OS with a weight of 0.78 was relatively very close. Note that while in the 15-puzzle the overhead per node of calculating the heuristic was very small, the h_{GBC} heuristic described above requires significant time. This reduces the relative overhead of maintaining two OPEN lists required by OS. This explains the improved runtime of OS in comparison with the 15-puzzle results. It is important to note that APTS does not use any parameter while both AWA* and OS are very sensitive to the weight of the heuristic.[6] This can be seen in the degrading results of AWA*-1.3 and OS-1.3.

Conclusion and Future Work

In this paper we introduced a best-first search algorithm, Potential search (PTS), which is specifically designed to solve a bound-cost search problems. PTS orders the nodes according to their potential. Several ways to estimate the potential of a node are described. Specifically, we use the relation between a given heuristic and the optimal cost to a develop a cost function that can order the OPEN node according to their potential, without actually calculating it. In addition, PTS can be modified to an anytime search algorithm variant, APTS. Empirical results show both PTS variants are very efficient and outperformed other algorithms in most settings of our experiments.

The main advantage of PTS over the other algorithms we tried is that it does not require parameter tuning (such as w in the WA*-based algorithms) and is thus much more robust across different instances. Other algorithms were shown to be very sensitive to the parameter used. In many cases, e.g., when the optimal solution is unknown, one would not be able to determine the correct value for the parameter for these algorithms.

Future work will investigate how to incorporate an estimate of the search effort until the desired solution is found, in addition to the potential of a node. For example, a node that is very close to a goal might be preferred to a node that has a slightly higher potential but is farther from a goal. In addition, we are currently pursuing the use of machine learning technique to learn the potential function, instead of the indirect potential calculation described in this paper.

[6] Although OS can be used with any inadmissible heuristic, finding an efficient aparametric inadmissible heuristic is challenging.

Acknowledgments

This research was supported by the Israeli Science Foundation (ISF) grant no. 305/09 to Ariel Felner. We thank Robert Holte and Wheeler Ruml for their helpful discussion on a preliminary version of this paper.

References

Barabasi, A. L., and Albert, R. 1999. Emergence of scaling in random networks. *Science* 286(5439):509–512.

Everett, M. G., and Borgatti, S. P. 1999. The centrality of groups and classes. *Journal of Mathematical Sociology* 23(3):181–201.

Felner, A.; Korf, R. E.; and Hanan, S. 2004. Additive pattern database heuristics. *J. Artif. Intell. Res. (JAIR)* 22:279–318.

Freeman, L. C. 1977. A set of measures of centrality based on betweenness. *Sociometry* 40(1):35–41.

Furcy, D., and Koenig, S. 2005. Limited discrepancy beam search. In *IJCAI*, 125–131.

Hansen, E. A., and Zhou, R. 2007. Anytime heuristic search. *J. Artif. Intell. Res. (JAIR)* 28:267–297.

Katz, M., and Domshlak, C. 2010. Optimal admissible composition of abstraction heuristics. *Artif. Intell.* 174(12-13):767–798.

Korf, R. E. 1985. Depth-first iterative-deepening: An optimal admissible treesearch. *Artif. Intell.* 27(1):97–109.

Larsen, B. J.; Burns, E.; Ruml, W.; and Holte, R. 2010. Searching without a heuristic: Efficient use of abstraction. In *AAAI*.

Likhachev, M.; Gordon, G. J.; and Thrun, S. 2003. ARA*: Anytime A* with provable bounds on sub-optimality. In *NIPS*.

Pearl, J. 1984. *Heuristics: Intelligent search strategies for computer problem solving*. Addison-Wesley Pub. Co., Inc.,Reading, MA.

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

Puzis, R.; Elovici, Y.; and Dolev, S. 2007. Finding the most prominent group in complex networks. *AI Commun.* 20(4):287–296.

Sturtevant, N. R.; Felner, A.; Barrer, M.; Schaeffer, J.; and Burch, N. 2009. Memory-based heuristics for explicit state spaces. In *IJCAI*, 609–614.

Thayer, J. T., and Ruml, W. 2008. Faster than weighted A*: An optimistic approach to bounded suboptimal search. In *ICAPS*, 355–362.

Zanarini, A., and Pesant, G. 2009. Solution counting algorithms for constraint-centered search heuristics. *Constraints* 14:392–413.

Zhang, W., and Korf, R. E. 1995. Performance of linear-space search algorithms. *Artificial Intelligence* 79:241–292.

Zilberstein, S. 1996. Using anytime algorithms in intelligent systems. *AI Magazine* 17(3):73–83.