Auction Algorithms for Path Planning, Network Transport, and Reinforcement Learning †

by

Dimitri Bertsekas ‡

Abstract

We consider some classical optimization problems in path planning and network transport, and we introduce new auction-based algorithms for their optimal and suboptimal solution. The algorithms are based on mathematical ideas that are related to competitive bidding by persons for objects and the attendant market equilibrium, which underlie auction processes. However, the starting point of our algorithms is different, namely weighted and unweighted path construction in directed graphs, rather than assignment of persons to objects. The new algorithms have several potential advantages over existing methods: they are empirically faster in some important contexts, such as max-flow, they are well-suited for on-line replanning, and they can be adapted to distributed asynchronous operation. Moreover, they allow arbitrary initial prices, without complementary slackness restrictions, and thus are better-suited to take advantage of reinforcement learning methods that use off-line training with data, as well as on-line training during real-time operation. The new algorithms may also find use in reinforcement learning contexts involving approximation, such as multistep lookahead and tree search schemes, and/or rollout algorithms.

† A periodically updated and expanded version of a paper posted at arXiv:22207.09588 on July 19, 2022.

‡ Fulton Professor of Computational Decision Making, School of Computing and Augmented Intelligence, Arizona State University, Tempe, AZ.

1. INTRODUCTION

In this paper, we introduce new auction algorithms for broad classes of network flow problems. Our proposed methodology aims to improve the efficiency and flexibility of existing auction algorithms for linear single commodity network optimization, including shortest path planning, matching, assignment, and network transportation problems. Our emphasis will be on shortest path problems, but the ideas apply more broadly, particularly since the shortest path problem finds application in many contexts beyond network optimization, such as reinforcement learning.

Auction algorithms for network flow optimization have a long history, starting with the author's original paper [Ber79] that dealt with the assignment problem. This algorithm is inspired by a process of economic competition through bidding upwards the prices of a number of objects by an equal number of persons. Auction algorithms are discussed in detail in many sources, including the author's book [Ber98] and tutorial survey [Ber92] (which contain many references, too numerous to list here). Besides excellent computational complexity properties, their advantages over other network flow methods include their suitability for reoptimization and parallelization. They have found use in contexts involving classical weighted matching, route planning, scheduling, and related network optimization problems.

Auction algorithms have also been considered widely in applications of optimal transport, which are currently very popular; see e.g., Brenier et al. [BFH03], Villani [Vil09], [Vil21], Santambrogio [San15], Galichon [Gal16], Schmitzer [Sch16], [Sch19], Walsh and Dieci [WaD17], [WaD19], Peyre and Cuturi [PeC19], Merigot and Thibert [MeT21], and the references quoted there. Several code implementations of auction algorithms have become publicly available, some of which are accessible from the author's website.

In this paper, we are also aiming at applications in machine learning, data mining, and artificial intelligence, taking advantage of the ability of our algorithms to adapt to changing environments, and their suitability for off-line and on-line training with data, through the use of machine learning and reinforcement learning techniques. For examples of related contexts, which have involved the use of auction algorithms, see the papers by Kosowsky and Yuille [KoY94], Bayati, Shah, and Sharma [BSS08], Jacobs, Merkurjev, and Esedoglu [JME18], Wang and Xia [WaX12], Lewis et al. [LBD21], Bicciato and Torsello]BiT22], and Clark et al. [CLG22].

Our primary focus in this paper is the classical problem of finding a shortest path from an origin node r to a destination node t in a directed graph, where each arc (i, j) has a given cost (or length) a_{ij}. Our proposed algorithms are largely new, but they are motivated by the auction algorithm for the assignment problem. A key idea is to convert the shortest path problem to an equivalent assignment problem, using a well-known transformation [Ber91]. This can be done using the device of node splitting, as illustrated with an example in Fig. 1.1: we split each node $i \neq r, t$ into two nodes, i and i', and connect them with an arc

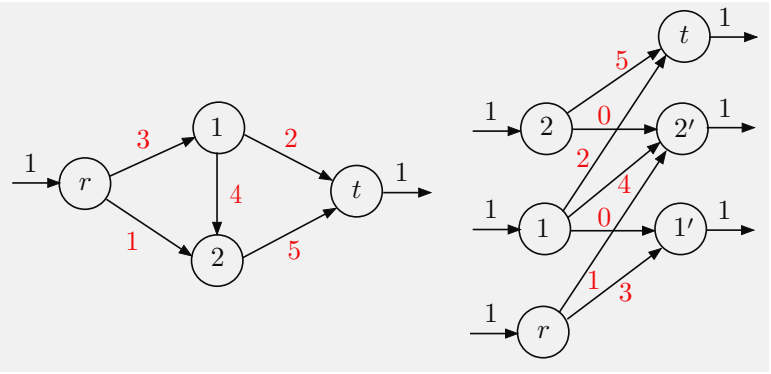


Figure 1.1. Transformation of a shortest path problem to an assignment problem from an origin node r to a destination node t. We use node splitting for the nodes 1 and 2. The arc costs are shown next to the arcs. For example, the path $(r, 1, 2, t)$ corresponds to the assignment

$$(r, 1'), (1, 2'), (2, t),$$

while the path $(r, 1, t)$ corresponds to the assignment $(r, 1'), (1, t), (2, 2')$ and the path $(r, 2, t)$ corresponds to the assignment $(r, 2'), (1, 1'), (2, t)$.

(i, i') of cost 0. Moreover, we replace the outgoing arcs (i, j) from all nodes $i \neq t$ with arcs (i, j'). Then the graph becomes bipartite and the problem is transformed to an assignment problem with persons representing the nodes $i \neq t$, objects representing the nodes j' and t, the costs $a_{ij'}$ being equal to the arc lengths a_{ij}, and the costs a_{it} being equal to the arc lengths a_{it}.

To establish the desired shortest path to assignment transformation, we note that any path from r to t of the form

$$(r, i_1, i_2, \ldots, i_k, t)$$

in the original shortest path problem corresponds to a feasible assignment, which contains the pairs

$$(r, i'_1), (i_1, i'_2), (i_2, i'_3), \ldots, (i_{k-1}, i'_k), (i_k, t), \tag{1.1}$$

and also possibly contains some additional pairs, such as the pairs (i, i') for $i \neq i_1, \ldots, i_k$, which are cost-free. In addition, we can see that the assignment problem has a feasible solution only if there exists at least one path from r to t, which we assume in our discussion.

Given a shortest path problem, one possibility is to consider the equivalent assignment problem, and to apply the auction/assignment algorithm with an initial partial assignment that consists of all the pairs (i, i'), $i \neq r, t$, along with node prices that satisfy ϵ-CS. Another possibility is to consider the equivalent assignment problem, and to simply start the auction algorithm with the empty assignment and arbitrary node prices. These two possibilities lead to somewhat different algorithms, but they both entail a very similar structure. In particular, the algorithms to be discussed in Sections 2 and 3 involve the idea of maintaining node prices and an acyclic path, which starts at the origin r and is iteratively extended or contracted by

adding a new node at the end of the path, or deleting the terminal node of the path. The reader may verify that the extension and contraction operations are related to the bidding operations of the auction algorithm, as applied to the equivalent assignment problem illustrated in Fig. 1.1.

In this paper, rather than focus on the details of the corresponding algorithmic relations and equivalences, we will simply adapt the auction process directly to the shortest path problem, taking advantage of its intuitive character. However, the shortest path to assignment transformation outlined above gives rise to a number of algorithmic variants, which are based on corresponding variants of the auction algorithm for the assignment problem.

The starting point for our development is an algorithm for computing some (not necessarily shortest) path connecting an origin node to a destination node in a directed graph. The algorithm uses node prices to guide the search for the path. A special case of this algorithm was given in the author's paper [Ber95a] as part of an auction algorithm for max-flow, and was also described in the book [Ber98], Section 3.3.1. The more general version proposed here allows unrestricted choice of the initial prices, which among others, facilitates its use in reinforcement learning and on-line replanning contexts.

Our algorithm, once generalized to apply to a shortest path problem, also resembles an auction/shortest path algorithm that was proposed in the author's paper [Ber91]; see also the book [Ber98], Section 2.6. In particular, both algorithms employ a contraction/extension mechanism for path construction. Contrary to this earlier algorithm, however, the algorithm of the present paper admits arbitrary initial prices, and uses a positive ϵ parameter to effect larger price changes, which in turn speeds up its convergence. It produces a shortest path for sufficiently small ϵ, and it is also well-suited for an ϵ-scaling approach, a central technique in auction algorithms, which improves computational efficiency. Another important difference is that the earlier algorithm has nonpolynomial complexity, whereas the algorithm of the present paper is polynomial thanks to the use of ϵ-scaling.†

The paper is organized as follows. In the next section we will consider a feasibility/path construction problem, involving a directed graph with an origin node s and a destination node t. We assume that there is at least one directed path from s to t, and we want to find one such path. One possibility is to use an algorithm that finds a shortest path from s to t, with respect to some set of arc lengths or weights; for example a length equal to 1 for every arc. There are several well-known algorithms to solve such a shortest path problem. However, in Section 2 we will introduce a simpler and faster algorithm that finds a path from s to t without regard to its optimality. In Section 3, we will discuss extensions of the path construction algorithm of Section 2, which use arc lengths and produce a path that is "nearly shortest" with respect to

† A polynomial variant of the 1991 auction algorithm, given in the paper [BPS95], performs very well for single origin-few destination problems with nonnegative arc lengths, but includes features that detract from the flexibility of our new algorithm of Section 3.

these lengths (and shortest under some conditions). In Section 4, we will discuss variations of the algorithms of Sections 2 and 3, including algorithms for shortest path problems with multiple origins and destinations, as well as distributed implementations. In Section 5, we will briefly discuss extensions of our path construction algorithms and their uses in solving matching, assignment, max-flow, transportation, and transhipment problems. More detailed discussions of these extensions is beyond the scope of the present paper and will be given in separate reports. Finally in Section 6, we will illustrate the application of auction/shortest path ideas to reinforcement learning contexts. More detailed discussions are again outside our scope and will be reported elsewhere.

2. PATH CONSTRUCTION IN A DIRECTED GRAPH

We will now introduce our auction algorithm for path construction. The algorithm finds just *some* path from origin to destination, without aiming for any kind of optimality properties. It maintains a path starting at the origin, which at each iteration, is either *extended* by adding a new node, or *contracted* by deleting its terminal node. The decision to extend or contract is based on a set of variables, one for each node, which are called *prices*. Roughly speaking, the price of a node is viewed as a measure of the desirability of revisiting and advancing from that node in the future (low-price nodes are viewed as more desirable). Once the destination becomes the terminal node of the path, the algorithm terminates.

To get an intuitive sense of the algorithm, think of a mouse moving in a graph-like maze, trying to reach the destination. The mouse criss-crosses the maze, either advancing or backtracking along its current path, guided by prices that encode how desirable the maze nodes/crosspoints are, based on the mouse's "learned" experience. The mouse advances forward from high price to low price nodes, going from a node to a downstream neighbor node only if that neighbor has lower price (or equal price under some conditions). It backtracks when it reaches a node whose downstream neighbors all have higher price. In this case, it also suitably increases the price of that node, thus marking the node as less desirable for future exploration, and providing an incentive to explore alternative paths to the destination.

Our algorithm emulates efficiently the search process just described, guided by a suitable set of rules for price updating. An important side benefit is that the prices provide the means to "transfer knowledge," in the sense that good learned prices from previous searches can be used as initial prices for subsequent related searches, with an attendant computational speedup.

2.1 Auction Path Construction Algorithm

We will now describe formally our algorithm, which we call *auction/path construction* (APC for short) for finding a path from the origin node s to the destination node t in a directed graph. The arcs of the graph

are denoted by (i, j), where i and j are referred to as the *start* and *end* nodes of the arc. The sets of nodes and the set of arcs are denoted by $\mathcal{N}$ and $\mathcal{A}$, respectively. If (i, j) is an arc, it is possible that (j, i) is also an arc. No self arcs of the form (i, i) are allowed. We assume that for any two nodes i and j, there is at most one arc with start node i and end node j. For any node i we say that node j is a *downstream neighbor of* i if (i, j) is an arc. A node i is called *deadend* if it has no downstream neighbors. Note that s is not deadend, since we have assumed that there is a path from s to t.

Our algorithm maintains and updates a scalar price p_i for each node i. We say that under the current set of prices an arc (i, j) is:

(a) *Downhill*: If $p_i > p_j$.

(b) *Level*: If $p_i = p_j$.

(c) *Uphill*: If $p_i < p_j$.

Our algorithm also maintains and updates a directed path $P = (s, n_1, \ldots, n_k)$ that starts at the origin, and is such that (s, n_1), and $(n_\ell, n_{\ell+1})$ with $\ell = 1, \ldots, k-1$, are arcs. The path is either the degenerate path $P = (s)$, or it ends at some node $n_k \neq s$, which is called the *terminal node* of P. If $P = (s)$, we also say that the terminal node of P is s.

Each iteration of the APC algorithm starts with a path and a price for each node, which are updated during the iteration using rules that we will now describe. The algorithm starts with the degenerate path $P = (s)$, and with some initial prices, which are arbitrary. † It terminates when a path has been found from s to t.

At each iteration when the algorithm starts with a path of the nondegenerate form $P = (s, n_1, \ldots, n_k)$, it either removes from P the terminal node n_k to obtain the new path $\overline{P} = (s, n_1, \ldots, n_{k-1})$, or it adds to P a node n_{k+1} to obtain the new path $\overline{P} = (s, n_1, \ldots, n_k, n_{k+1})$. In the former case the operation is called a *contraction* to n_{k-1}, and in the latter case it is called an *extension* to n_{k+1}.

At any one iteration the algorithm starts with a path P and a price p_i for each node i. At the end of the iteration a new path $\overline{P}$ is obtained from P through a contraction or an extension. Also the price of the terminal node of P [or the price p_s if $P = (s)$] is increased by a certain amount when there is a contraction. For iterations where the algorithm starts with the degenerate path $P = (s)$, only an extension is possible,

† The arbitrary nature of the initial prices is a major difference of our algorithm from the earlier auction/path construction algorithms given in [Ber98], Section 2.6 and 3.3. Allowing arbitrary initial prices allows more flexibility in reusing prices from solution of one path finding problem to another similar problem. It also facilitates the use of "learned" prices that are favorable in similar problem contexts. This property can be important for computational efficiency in many applications.

i.e., $P = (s)$ is replaced by a path of the form $\overline{P} = (s, n_1)$.

A key feature of the algorithm, which in fact motivates its design, is that P and the prices p_i satisfy the following property at the start of each iteration for which $P \neq (s)$.

Downhill Path Property:

All arcs of the path $P = (s, n_1, \ldots, n_k)$ maintained by the APC algorithm are level or downhill. Moreover, the last arc (n_{k-1}, n_k) of P is downhill following an extension to n_k.

The significance of the downhill path property is that *when an extension occurs, a cycle cannot be created*, in the sense that the terminal node n_k is different than all the predecessor nodes $s, n_1, \ldots, n_{k-1}$ on the path P. The reason is that the downhill path property implies that following an extension, we must have

$$p_{n_k} < p_{n_{k-1}} \leq p_{n_{k-2}} \leq \cdots \leq p_{n_1} \leq p_s,$$

showing that the terminal node n_k following an extension cannot be equal to any of the preceding nodes of P.

In addition to maintaining the downhill path property, the algorithm is structured so that following a contraction, which changes a nondegenerate path of the form $P = (s, n_1, \ldots, n_k)$ to $\overline{P} = (s, n_1, \ldots, n_{k-1})$, the price of n_k is increased by a positive amount. In conjunction with the fact that P never contains a cycle, this implies that either the algorithm terminates, or some node prices will increase to infinity. This is the key idea that underlies the validity of the algorithm, and forms the basis for its proof of termination.

To describe formally the algorithm, consider the case where $P \neq (s)$ and P has the form $P = (s, n_1, \ldots, n_k)$. We then denote by

$$\text{pred}(n_k) = n_{k-1}$$

the predecessor node of the terminal node n_k in the path P. [In the case where $P = (s, n_1)$, we use the notation $\text{pred}(n_1) = s$.] If the terminal node n_k of P is not deadend, we denote by $\text{succ}(n_k)$ a downstream neighbor of n_k that has minimal price:

$$\text{succ}(n_k) \in \arg \min_{\{j \mid (n_k, j) \in \mathcal{A}\}} p_j.$$

If multiple downstream neighbors of n_k have minimal price, the algorithm designates arbitrarily one of these neighbors as $\text{succ}(n_k)$.

The algorithm also uses a positive scalar ϵ. The choice of ϵ does not affect the path produced by the algorithm (so we could use $\epsilon = 1$ for example), but the choice of ϵ will play an important role in the weighted path construction algorithm of the next section.

The rules by which the path P and the prices p_i are updated at each iteration are as follows; see Fig. 2.1.

Auction Algorithm Iteration for Unweighted Path Construction: We distinguish three mutually exclusive cases.

(a) $P = (s)$: We then set the price p_s to $\max\{p_s, p_{\text{succ}(s)} + \epsilon\}$, and extend P to $\text{succ}(s)$.

(b) $P = (s, n_1, \ldots, n_k)$ *and node* n_k *is deadend*: We then set the price p_{n_k} to ∞ (or a very high number for practical purposes), and contract P to n_{k-1}.

(c) $P = (s, n_1, \ldots, n_k)$ *and node* n_k *is not deadend*. We consider the following two cases.

(1) $p_{\text{pred}(n_k)} > p_{\text{succ}(n_k)}$. We then extend P to $\text{succ}(n_k)$ and set p_{n_k} to any price level that makes the arc $\big(\text{pred}(n_k), n_k\big)$ level or downhill and the arc $\big(n_k, \text{succ}(n_k)\big)$ downhill. [Setting

$$p_{n_k} = p_{\text{pred}(n_k)},$$

which raises p_{n_k} to the maximum possible level, is a possibility. In this case the arc $\big(\text{pred}(n_k), n_k\big)$ becomes level; see Fig. 2.2.]

(2) $p_{\text{pred}(n_k)} \leq p_{\text{succ}(n_k)}$. We then contract P to $\text{pred}(n_k)$ and raise the price of n_k to the price of $\text{succ}(n_k)$ plus ϵ [thus making the arc $\big(\text{pred}(n_k), n_k\big)$ uphill and the arc $\big(n_k, \text{succ}(n_k)\big)$ downhill].

The algorithm terminates once the destination becomes the terminal node of P. We will show that eventually the algorithm terminates, under our standing assumption that there is at least one path from the origin to the destination.

The contraction/extension mechanism of the algorithm may be interpreted as a competitive process: we can view $\text{pred}(n_k)$ as being in competition with the downstream nodes of n_k for becoming the next terminal node of path P, after n_k. In particular, the terminal node of P moves to the node that offers minimal price [with ties that involve $\text{pred}(n_k)$ broken in favor of $\text{pred}(n_k)$ in order to maintain the downhill path property].

Figure 2.1 illustrates the extension and contraction mechanism of case (c) above, and shows how the downhill path property of the algorithm is maintained throughout its operation. In particular, the initial path $P = (s)$ satisfies the downhill path property trivially, since it contains no arcs. Furthermore, using Fig. 2.1 and the algorithm description, we can verify that if P and the node prices satisfy the downhill path property at the beginning of an iteration, then the new path and node prices at the beginning of the next iteration also satisfy the downhill path property. Figure 2.2 illustrates the extension and contraction mechanism in the special case where p_{n_k} is raised to the maximum possible level $p_{n_k} = p_{\text{pred}(n_k)}$ following

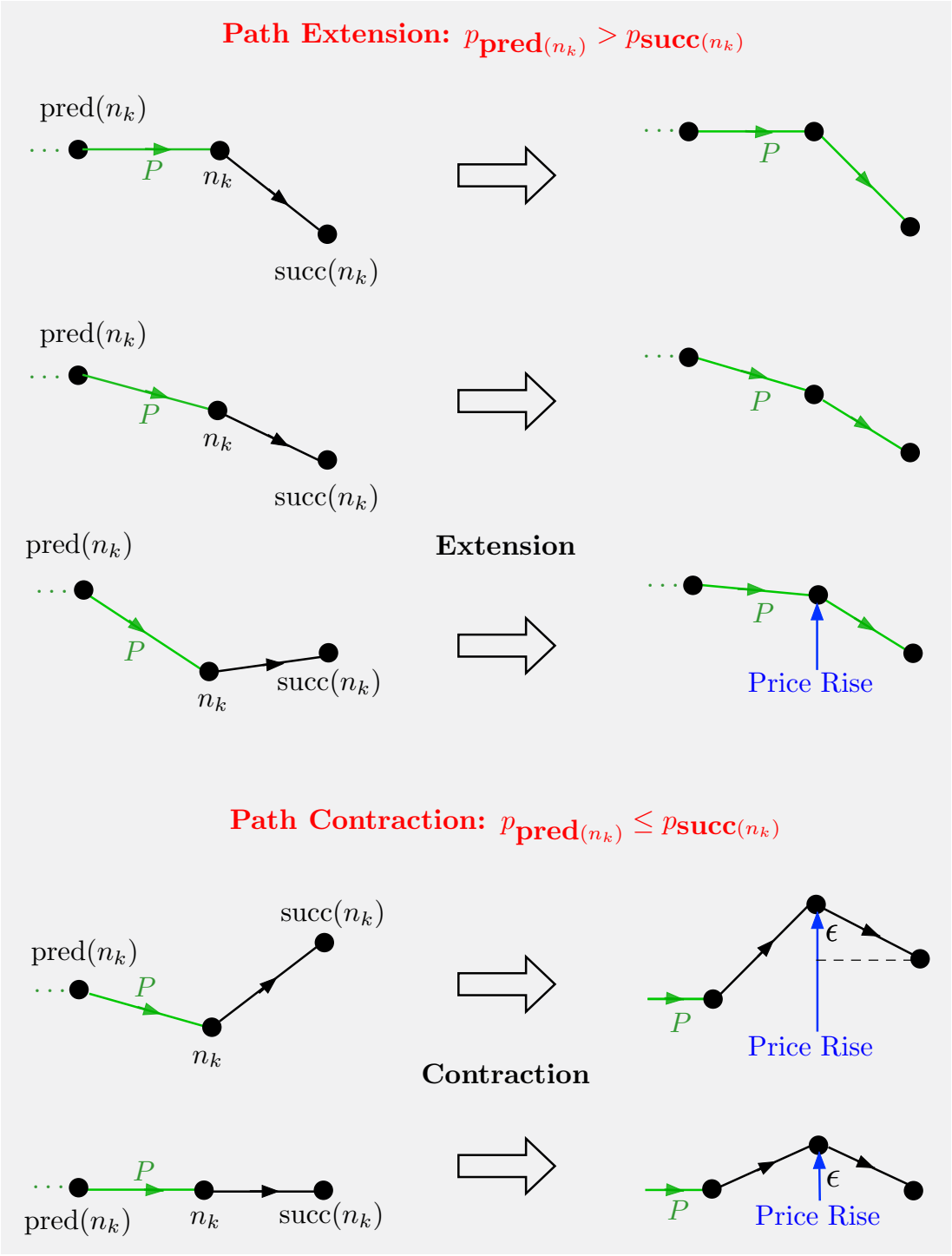


Figure 2.1 Illustration of the price levels of the terminal node n_k of the path $P = (s, n_1, \ldots, n_k)$, and the price levels of its predecessor and its successor, before and after an extension or a contraction; cf. cases (c1) and (c2) of the algorithm description. In the case where $p_{\text{pred}(n_k)} > p_{\text{succ}(n_k)}$, which corresponds to an extension, there may or may not be an increase of p_{n_k}. In the case where $p_{\text{pred}(n_k)} \leq p_{\text{succ}(n_k)}$, which corresponds to a contraction, there is always an increase of p_{n_k} by at least ϵ.

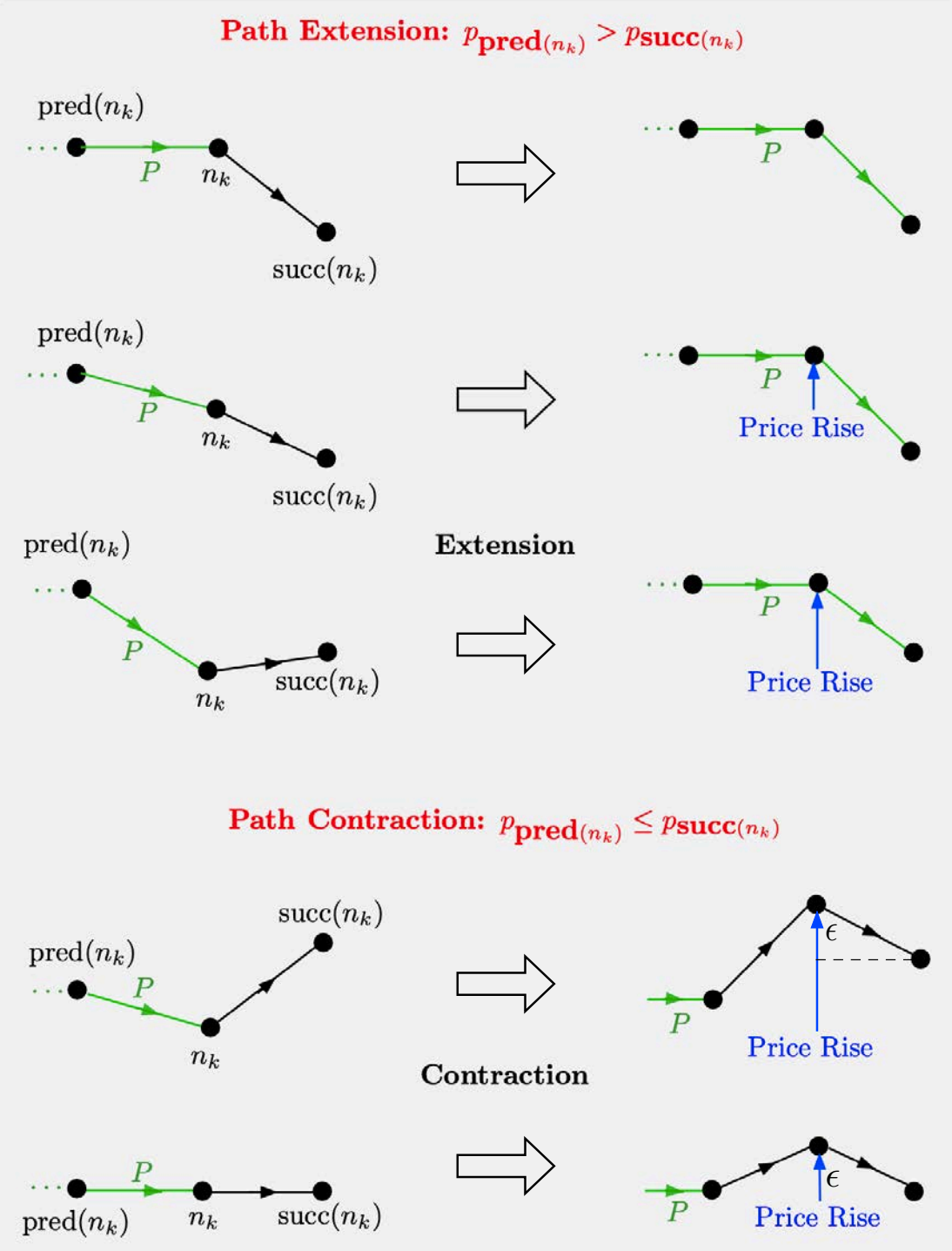


Figure 2.2 Illustration of the special case of the APC algorithm that sets $p_{n_k} = p_{\text{pred}(n_k)}$, raising p_{n_k} to the maximum possible level $p_{n_k} = p_{\text{pred}(n_k)}$ in the case of the extension step (c1). In this case the predecessor arc $\big(\text{pred}(n_k), n_k\big)$ is forced to become level, and the arcs of path P are all level, except for the last arc following an extension.

an extension, thus making the predecessor arc $\big(\text{pred}(n_k), n_k\big)$ level.

2.2 Algorithm Justification

We will prove that eventually the destination will become the terminal node of P, at which time the algorithm will terminate. To this end we argue by contradiction and we use our assumption that there is at least one path from the origin to the destination.

Indeed, suppose, to arrive at a contradiction, that the algorithm does not terminate. Then, since the path P does not contain a cycle and hence cannot extend indefinitely, the algorithm must perform an infinite number of contractions. Let $\mathcal{N}_\infty$ be the nonempty set of nodes whose price increases (by at least ϵ) infinitely often due to a contraction (and hence their price increases to ∞). Let also $\overline{\mathcal{N}}_\infty = \{i \mid i \notin \mathcal{N}_\infty\}$ be the complementary set of nodes whose price increases due to a contraction finitely often (and hence do not become the terminal node of P after some iteration). Clearly, by the rules of the algorithm, there is no arc connecting a node of $\mathcal{N}_\infty$ to a node of $\overline{\mathcal{N}}_\infty$. Moreover, the destination t clearly belongs to $\overline{\mathcal{N}}_\infty$, and we claim that the origin s belongs to $\mathcal{N}_\infty$. Indeed, if $s \in \overline{\mathcal{N}}_\infty$ there would exist a subpath $P' = (s, n_1, \ldots, n_k)$ such that the nodes $s, n_1, \ldots, n_{k-1}$ belong to $\overline{\mathcal{N}}_\infty$, the last node n_k belongs to $\mathcal{N}_\infty$, and P' is the initial portion of P for all iterations after finitely many. Since n_k will be the terminal node of P infinitely often, it follows that n_{k-1} will be the predecessor $\text{pred}(n_k)$ of n_k infinitely often, while the price of n_k increases to infinity and the price of n_{k-1} stays finite. By the rules of the algorithm, this is not possible. Thus we must have $s \in \mathcal{N}_\infty$, $t \in \overline{\mathcal{N}}_\infty$, and no arc connecting a node of $\mathcal{N}_\infty$ to a node of $\overline{\mathcal{N}}_\infty$. This contradicts the assumption that there is a path from s to t, and shows that the algorithm will terminate.

We summarize the preceding arguments in the following proposition.

Proposition 2.1: If there exists at least one path from the origin to the destination, the APC algorithm terminates with a path from s to t. Otherwise the algorithm never terminates and we have $p_i \to \infty$ for all nodes i in a subset $\mathcal{N}_\infty$ that contains s.

3. PATH PLANNING WITH ARC WEIGHTS

We will now introduce a generalization of our path construction algorithm, which we call *auction/weighted path construction* (AWPC for short). The algorithm incorporates a length (or weight) a_{ij} for every arc (i, j), and aims to provide a path with near-minimum total length. Each length a_{ij} encodes a measure of desirability of including arc (i, j) into a path from the origin to the destination. The arc lengths serve to provide a bias towards producing paths with small total length. In fact in many cases (but not always)

the algorithm produces shortest paths with respect to the given lengths. We require that *all cycles have nonnegative length*. By this we mean that for every cycle $(i, n_1, \ldots, n_k, i)$ we have

$$a_{i,n_1} + a_{n_1 n_2} + \cdots + a_{n_{k-1} n_k} + a_{n_k i} \geq 0. \tag{3.1}$$

This is a common assumption in shortest path problems.†

3.1 The AWPC Algorithm

The AWPC algorithm maintains and updates a directed path $P = (s, n_1, \ldots, n_k)$ and a price p_i for each node i. Extending the terminology of the preceding section, we say that under the current set of prices and lengths an arc (i, j) is:

(a) *Downhill*: If $p_i > a_{ij} + p_j$.

(b) *Level*: If $p_i = a_{ij} + p_j$.

(c) *Uphill*: If $p_i < a_{ij} + p_j$.

As earlier, we denote by

$$\text{pred}(n_k) = n_{k-1}$$

the predecessor node of the terminal node n_k in the path P. [In the case where $P = (s, n_1)$, we let $\text{pred}(n_1) = s$.] If the terminal node n_k of P is not deadend, we denote by $\text{succ}(n_k)$ a downstream neighbor j of n_k for which $a_{n_k j} + p_j$ is minimized:

$$\text{succ}(n_k) \in \arg \min_{\{j \mid (n_k, j) \in \mathcal{A}\}} \{a_{n_k j} + p_j\}.$$

† The earlier auction/shortest path algorithm of [Ber91] requires that all cycle lengths be strictly positive rather than nonnegative, which is often a significant restriction. Another important difference, which affects computational efficiency, is that there is no ϵ parameter in the algorithm of [Ber91]. Indeed, this algorithm is closely related to the so called "naive auction algorithm," which is the auction algorithm for the assignment problem with $\epsilon = 0$; see [Ber91]. The AWPC algorithm, to be presented shortly, can also be operated with $\epsilon = 0$, assuming that all cycle lengths are strictly positive. This requires a small change: in the case (c2) where $p_{\text{pred}(n_k)} = p_{\text{succ}(n_k)}$, we do an extension while setting $p_{n_k} = p_{\text{pred}(n_k)} = p_{\text{succ}(n_k)}$, rather than doing a contraction. Then the path P may not contain any uphill arcs, but it consists of just level arcs during operation of the algorithm. However, still the critical property that a cycle cannot be created through an extension is preserved, under the strict cycle length positivity assumption. Introducing a positive parameter ϵ allows for nonnegative cycle lengths, and also provides a mechanism for controlling the rate of convergence of the algorithm through the technique of ϵ-scaling, as will be discussed later in this section.

If multiple downstream neighbors of n_k attain the minimum, the algorithm designates arbitrarily one of these neighbors as $\text{succ}(n_k)$.

Note that when $a_{ij} = 0$ for all arcs (i, j), the preceding definitions coincide with the ones given in the preceding section. Indeed when $a_{ij} = 0$, the AWPC algorithm to be presented shortly coincides with the APC algorithm of Section 2.

The AWPC algorithm maintains a directed path $P = (s, n_1, \ldots, n_k)$ that starts at the origin and consists of distinct nodes. The path is either the degenerate path $P = (s)$, or it ends at some node $n_k \neq s$, which, as earlier, is called the *terminal node* of P. Each iteration starts with a path and a price for each node, which are updated during the iteration. The algorithm starts with the degenerate path $P = (s)$, and the initial prices are arbitrary. It terminates when the destination becomes the terminal node of P.

We will now describe the rules by which the path and the prices are updated. At any one iteration the algorithm starts with a path P and a scalar price p_i for each node i. At the end of the iteration a new path $\overline{P}$ is obtained from P through a contraction or an extension as earlier. For iterations where the algorithm starts with the degenerate path $P = (s)$, only an extension is possible, i.e., $P = (s)$ is replaced by a path of the form $\overline{P} = (s, n_1)$. Also the price of the terminal node of P is increased just before a contraction, and in some cases, just before an extension. The amount of price rise is determined by a scalar parameter $\epsilon > 0$.

The algorithm terminates when the destination becomes the terminal node of P. The rules by which the path P and the prices p_i are updated at every iteration prior to termination are as follows.

Auction Algorithm Iteration for Weighted Path Construction: We distinguish three mutually exclusive cases.

(a) $P = (s)$: We then set the price p_s to $\max\{p_s,\, a_{s\text{succ}(s)} + p_{\text{succ}(s)} + \epsilon\}$, and extend P to $\text{succ}(s)$.

(b) $P = (s, n_1, \ldots, n_k)$ *and node* n_k *is deadend*: We then set the price p_{n_k} to ∞ (or a very high number for practical purposes), and contract P to n_{k-1}.

(c) $P = (s, n_1, \ldots, n_k)$ *and node* n_k *is not deadend*. We consider the following two cases.

(1) $p_{\text{pred}(n_k)} > a_{\text{pred}(n_k)n_k} + a_{n_k\text{succ}(n_k)} + p_{\text{succ}(n_k)}$. We then extend P to $\text{succ}(n_k)$ and set p_{n_k} to any price level that makes the arc $\big(\text{pred}(n_k), n_k\big)$ level or downhill and the arc $\big(n_k, \text{succ}(n_k)\big)$ downhill. [Setting

$$p_{n_k} = p_{\text{pred}(n_k)} - a_{\text{pred}(n_k)n_k},$$

thus raising p_{n_k} to the maximum possible level, is a possibility. In this case the arc $\big(\text{pred}(n_k), n_k\big)$ becomes level; cf. Fig. 2.2.]

(2) $p_{\text{pred}(n_k)} \leq a_{\text{pred}(n_k)n_k} + a_{n_k\text{succ}(n_k)} + p_{\text{succ}(n_k)}$. We then contract P to $\text{pred}(n_k)$ and raise the price of n_k to

$$a_{n_k\text{succ}(n_k)} + p_{\text{succ}(n_k)} + \epsilon$$

[thus making the arc $\big(\text{pred}(n_k), n_k\big)$ uphill and the arc $\big(n_k, \text{succ}(n_k)\big)$ downhill].

A downhill/level/uphill type of interpretation, similar to Fig. 2.1, applies to this algorithm as well [the relative heights of the prices of nodes $\text{pred}(n_k)$, $\text{succ}(n_k)$, and n_k, indicated in Fig. 2.1 should incorporate the arc lengths $a_{\text{pred}(n_k)n_k}$ and $a_{n_k\text{succ}(n_k)}$, as in the preceding algorithm description]; see Fig. 3.1. There is a price increase of n_k in the case of a contraction, and also in the case of an extension if the arc $\big(\text{pred}(n_k), n_k\big)$ is downhill. However, the conditions for an arc (i, j) to be downhill, level, or uphill involve the arc lengths a_{ij}. A key property that can be easily verified is that P and the prices p_i satisfy the following downhill path property at the start of each iteration for which $P \neq (s)$.

Downhill Path Property:

All arcs of the path $P = (s, n_1, \ldots, n_k)$ maintained by the AWPC algorithm are level or downhill. Moreover, the last arc (n_{k-1}, n_k) of P is downhill following an extension to n_k.

A consequence of this property (and our assumption that all cycles have nonnegative length) is that when an extension occurs, a cycle cannot be created, in the sense that the terminal node n_k is different than all the predecessor nodes $s, n_1, \ldots, n_{k-1}$ on the path $P = (s, n_1, \ldots, n_k)$. Thus, assuming that there is at least one path from the origin to the destination, it can be shown that eventually the destination will become the terminal node of P, at which time the algorithm will terminate. The proof is essentially identical to the proof we gave earlier for the case of zero lengths in Prop. 2.1. Figure 3.2 illustrates the algorithm with an example.

Proposition 3.1: If there exists at least one path from the origin to the destination and the nonnegative cycle condition (3.1) holds, the AWPC algorithm terminates with a path from s to t. Otherwise the algorithm never terminates and we have $p_i \to \infty$ for all nodes i in a subset $\mathcal{N}_\infty$ that contains s.

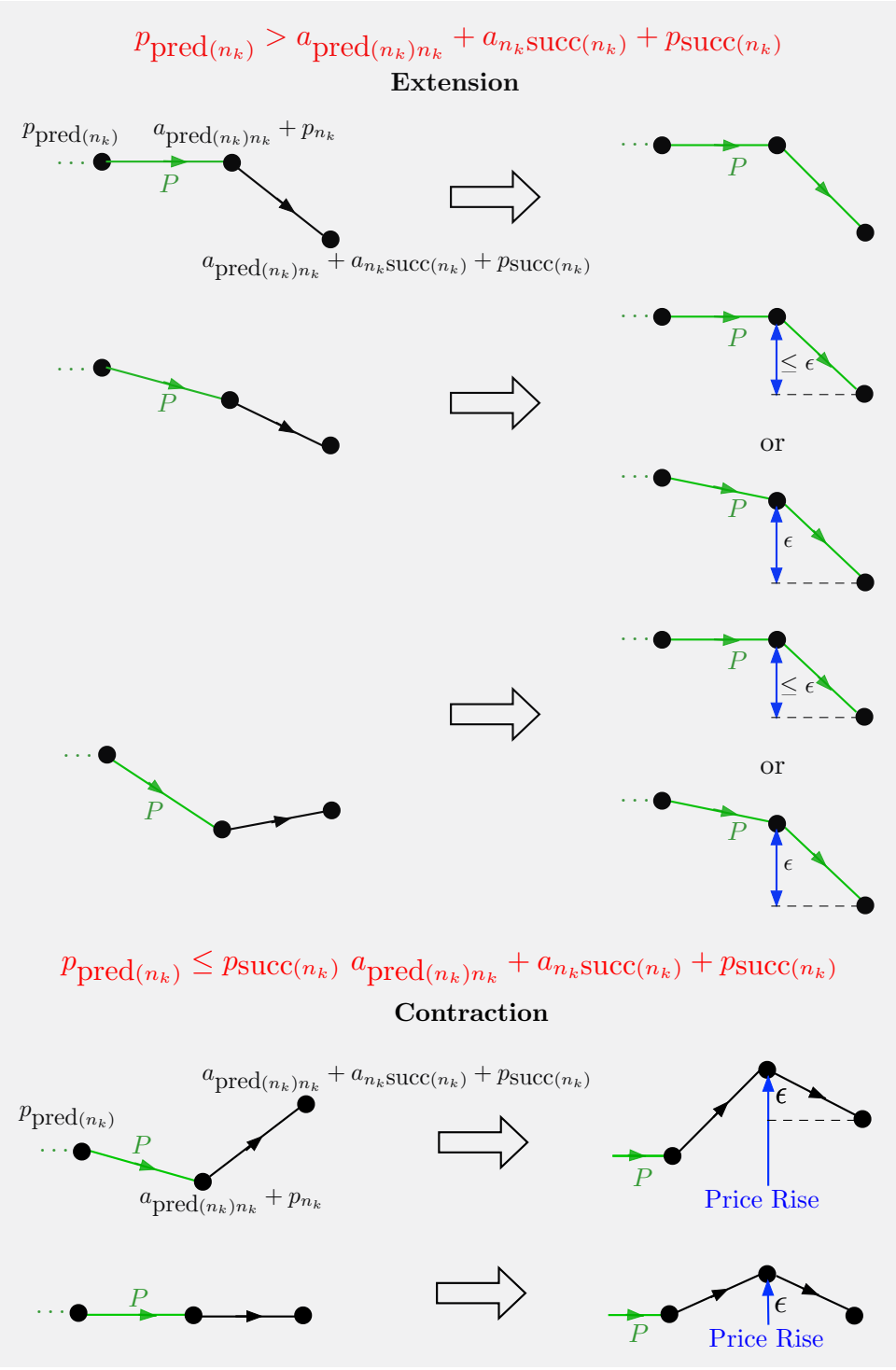
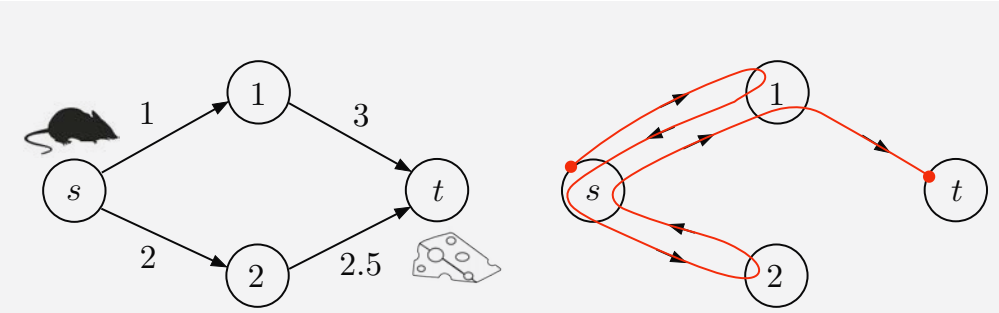


Figure 3.1 Illustration of an iteration of the ASP algorithm for the case where $P = (s, n_1, \ldots, n_k)$ with $n_k \neq s$. The figure shows the levels

$$p_{\text{pred}(n_k)}, \quad a_{\text{pred}(n_k)n_k} + p_{n_k}, \quad a_{\text{pred}(n_k)n_k} + a_{n_k\text{succ}(n_k)} + p_{\text{succ}(n_k)},$$

before and after an extension (top figure), and before and after a contraction (bottom figure).



Iteration #	**Path P prior to iteration**	**Price vector (p_s, p_1, p_2, p_t) prior to iteration**	**Type of iteration**
1	(s)	$(\underline{0}, 0, 0, 0)$	Extension to 1
2	$(s, 1)$	$(1+\epsilon, \underline{0}, 0, 0)$	Contraction to s
3	(s)	$(\underline{1+\epsilon}, 3+\epsilon, 0, 0)$	Extension to 2
4	$(s, 2)$	$(2+\epsilon, 3+\epsilon, \underline{0}, 0)$	Contraction to s
5	(s)	$(\underline{2+\epsilon}, 3+\epsilon, 2.5+\epsilon, 0)$	Extension to 1
6	$(s, 1)$	$(4+2\epsilon, \underline{3+\epsilon}, 2.5+\epsilon, 0)$	Extension to t
7	$(s, 1, t)$	$(4+2\epsilon, 3+2\epsilon, 2.5+\epsilon, \underline{0})$	Termination

Figure 3.2: A four-node example, with the arc lengths shown next to the arcs in the left-side figure. Roughly speaking, the algorithm first moves greedily to node 1, then contacts back to s upon encountering the high cost arc $(1, t)$. It then explores the alternative of going to node 2, and then returns through s and 1, to reach the destination.

The right-side figure and the table trace the steps of the AWPC algorithm starting with $P = (s)$ and all initial prices equal to 0. The algorithm is operated so that in the extension case (c1), we raise the price level of n_k to the maximum possible level, which is $p_{\text{pred}(n_k)} - a_{\text{pred}(n_k)n_k}$. The price of the terminal node of the path is underlined. The trajectory of the AWPC algorithm shown in the table corresponds to values of $\epsilon \leq 3$. The final path obtained is the shortest path $(s, 1, t)$. If instead $\epsilon > 3$, the algorithm will still find the shortest path and faster: it will first perform an extension to 1, setting $p_s = 1 + \epsilon$. It will then perform an extension to t, since the condition

$$1 + \epsilon = p_{\text{pred}(1)} > a_{\text{pred}(1)1} + a_{1\text{succ}(1)} + p_{\text{succ}(1)} = 1 + 3 + 0 = 4$$

is satisfied, and terminate. The final path will be $P = (s, 1, t)$ and the final price vector will be

$$(p_s, p_1, p_2, p_t) = (1 + \epsilon, 1 + \epsilon, 0, 0),$$

with the arc $(s, 1)$ being balanced and the arc $(1, t)$ being downhill.

Note also that generally, there is no guarantee that the AWPC algorithm will find a shortest path for all initial prices and values of ϵ. However, it will find a shortest path if ϵ is sufficiently small, provided the initial prices satisfy a certain ϵ-complementary slackness constraint that will be given in Section 3.4.

3.2 The Role of the Parameter ϵ - Convergence Rate and Solution Accuracy Tradeoff

In auction algorithms, it is common to use a positive ϵ parameter to regulate the size of price rises. In the AWPC algorithm, ϵ is used to provide an important tradeoff between the ability of the algorithm to construct paths with near-minimum length, and its rate of convergence. Generally, as ϵ becomes smaller the quality of the path produced improves, as we will show with examples and analysis in what follows. On the other hand a small value of ϵ tends to slow down the algorithm.

In what follows in this section, we will use two examples to illustrate how the choice of ϵ affects the rate of convergence of the AWPC algorithm, as well as the error from optimality of the path that it produces. For both examples, in the extension case (c1), we set the price level of n_k to the maximum possible level,

$$p_{n_k} = p_{\text{pred}(n_k)} - a_{\text{pred}(n_k)n_k},$$

cf. Fig. 2.2.

Example 3.1 (Nonpolynomial Behavior and ϵ-Scaling)

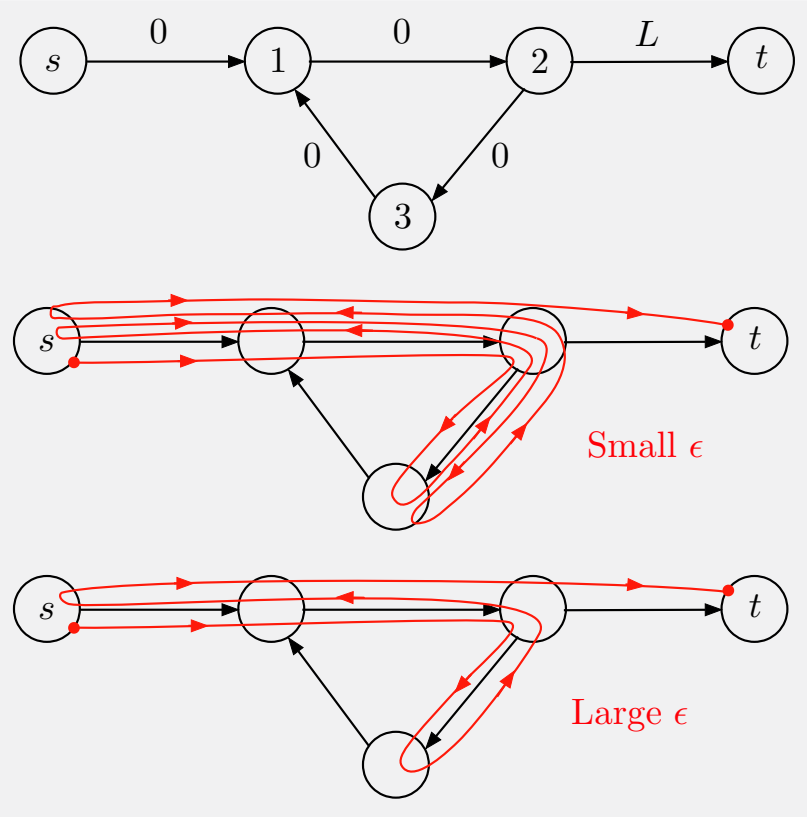
This is an example of a shortest path problem where there is a cycle of relatively small length. It involves that graph of the top figure in Fig. 3.3. The cycle consists of nodes 1, 2, and 3, and has length 0 [the algorithm's behavior is similar when the cycle has positive length that is small relative to L, the length of the last arc of the unique s-to-t path]. Such cycles slow down the algorithm, when ϵ has a small value. Indeed, it can be seen from the table of Fig. 3.3 that for small values of ϵ and initial prices equal to 0, the algorithm repeats the cycle

$$s \to 1 \to 2 \to 3 \to 2 \to 1 \to s \to 1 \cdots$$

until the prices of nodes 2 and 3 reach levels $p_2 > L$ and $p_3 > L$, so that the arc $(2, t)$ becomes downhill and an extension from 2 to t is performed. The number of cycles for this to happen depends on ϵ and is roughly proportional to L/ϵ, so for small values of ϵ, the computation is nonpolynomial (see the middle part of the figure). On the other hand, it can be seen from the table of Fig. 3.3 that when ϵ is large enough so that $3 + 4\epsilon > L$, the algorithm moves to t once it reaches node 2 for the second time, after 8 iterations.

It can be shown that with an ϵ-scaling scheme, whereby ϵ is reduced by a certain factor between successive runs of the algorithm, the computation becomes polynomial, proportional to $\log L$, rather than L. This is a common property of auction algorithms. A related complexity analysis is given in the paper [Ber91]; see also the book [Ber98] and the references quoted there. Later we will discuss the use of ϵ-scaling and its use to provide convergence acceleration as well as exact shortest path solutions.

In the preceding example, the poor performance of the algorithm is caused by the presence of a cycle with small length. The next example illustrates how a similar phenomenon can also occur in acyclic graphs involving many-node paths.



Iteration #	Path P prior to iteration	Price vector $(p_s, p_1, p_2, p_3, p_t)$ prior to iteration	Type of iteration
1	(s)	$(\underline{0}, 0, 0, 0, 0)$	Extension to 1
2	$(s, 1)$	$(\epsilon, \underline{0}, 0, 0, 0)$	Extension to 2
3	$(s, 1, 2)$	$(\epsilon, \epsilon, \underline{0}, 0, 0)$	Extension to 3
4	$(s, 1, 2, 3)$	$(\epsilon, \epsilon, \epsilon, \underline{0}, 0)$	Contraction to 2
5	$(s, 1, 2)$	$(\epsilon, \epsilon, \underline{\epsilon}, 2\epsilon, 0)$	Contraction to 1
6	$(s, 1)$	$(\epsilon, \underline{\epsilon}, 3\epsilon, 2\epsilon, 0)$	Contraction to s
7	(s)	$(\underline{\epsilon}, 4\epsilon, 3\epsilon, 2\epsilon, 0)$	Extension to 1
8	$(s, 1)$	$(5\epsilon, \underline{4\epsilon}, 3\epsilon, 2\epsilon, 0)$	Extension to 2
9	$(s, 1, 2)$	$(5\epsilon, 5\epsilon, \underline{3\epsilon}, 2\epsilon, 0)$	Extension to t if $2\epsilon > L$
			Extension to 2 otherwise
...	...	...	Continue until $p_3 > L$

Figure 3.3 The shortest path problem of Example 3.1 (top part of the figure). The arc lengths are shown next to the arcs [all lengths are equal to 0, except for the length of arc $(2, t)$ which has a large length L]. There is only one point where the algorithm can go wrong, at node 2 where there is a choice between going to t or going to 3. The only s-to-t path is $(s, 1, 2, t)$, but if ϵ is very small, the algorithm explores the possibility of reaching the destination through node 3 for many iterations, while repeating the cycle $s \to 1 \to 2 \to 3 \to 2 \to 1 \to s \to 1 \ldots$ (middle part of the figure). On the other hand, if $2\epsilon > L$, then at iteration 9, following an extension to node 2, the successor to node 2 is t, the algorithm compares the prices of nodes 3 and t, performs an extension to t, and terminates (bottom part of the figure).

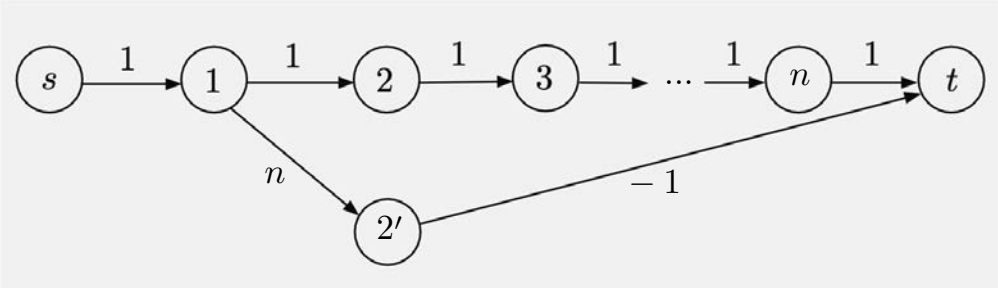


Figure 3.4 The graph of the shortest path problem of Example 3.2. The arc lengths are shown next to the arcs. All lengths are equal to 1, except for the length of arc $(2', t)$ which is equal to -1 and the length of arc $(1, 2')$ which is equal to n, the number of intermediate nodes in the top path from s to t. The shortest path is $(s, 2', t)$, but for large values of ϵ, the algorithm will terminate with the suboptimal path $(s, 1, 2, \ldots, n, t)$.

Example 3.2 (Convergence Rate and Solution Accuracy Tradeoff)

Consider a graph involving a long chain of nodes that starts at the origin and ends at the destination, as shown in Fig. 3.4. We assume that the initial prices are all equal to 0. Then it can be verified that for large values of ϵ, the algorithm will terminate with the suboptimal path $(s, 1, 2, \ldots, n, t)$; in fact for $\epsilon > n$, it will terminate in $n + 1$ iterations through the sequence of extensions $s \to 1 \to 2 \to \cdots \to n \to t$. It can also be verified, by tracing the steps of the algorithm that for small values of ϵ, the algorithm will find the optimal path $(s, 2', t)$, but will need a large number of iterations (proportional to n^2) to do so. A suitable ϵ-scaling scheme can find the optimal path in $O(n \log n)$ iterations.

3.3 Shortest Distances and Error Bounds

An important issue in the AWPC algorithm is the choice of the initial prices. We will argue that the algorithm operates effectively, in the sense that it terminates fast and with small error from path optimality, if the initial prices are close to the shortest distances under the given set of lengths. Indeed, the lengths $\{a_{ij}\}$ define the shortest distances, denoted by D_i^*, from the nodes i to the destination t. These shortest distances satisfy $D_t^* = 0$ and for all $i \neq t$,

$$D_i^* = \min_{\{j|(i,j)\in\mathcal{A}\}} \{a_{ij} + D_j^*\};$$

this is an instance of the fundamental dynamic programming/Bellman equation. It implies that all arcs are level or uphill, with respect to prices $p_i = D_i^*$, and the arcs of a shortest path are level. Suppose that we choose for all i a price p_i that is exactly equal to D_i^*. Then it can be verified that starting from an arbitrary origin i, the algorithm generates a shortest path from i to t through a sequence of extensions over level arcs, without any intervening contractions. The price differential $p_s - p_t$ is equal to the total length of the path produced by the algorithm, which is shortest.

If the initial prices p_i are not equal to the shortest distances D_i^*, the price differential $p_s - p_t$ provides an upper bound to the total length L_P of the final path P produced by the algorithm:

$$L_P = \sum_{(i,j)\in P} a_{ij} \leq p_s - p_t. \tag{3.2}$$

To see this, note that for $P = (s, n_1, \ldots, n_k)$ we have $a_{sn_1} = p_s - p_{n_1}$, $a_{n_{i-1}n_i} = p_{n_{i-1}} - p_{n_i}$ for all $i = 1, \ldots, k$, and $a_{n_k t} \leq p_{n_k} - p_t$, since in view of the downhill property, all the arcs of P are level or downhill. It follows that

$$L_P = a_{sn_1} + a_{n_1n_2} + \cdots + a_{n_{k-1}n_k} + a_{n_k t} \leq (p_s - p_{n_1}) + (p_{n_1} - p_{n_2}) + \cdots + (p_{n_{k-1}} - p_{n_k}) + (p_{n_k} - p_t) = p_s - p_t,$$

thus verifying Eq. (3.2).

If all the arcs that do not belong to P are level or uphill upon termination, then it can be shown that P has minimum total length. More generally, we will show an error bound that involves the amounts d_{ij} by which a_{ij} must be increased to make the arc (i, j) level if it is downhill:

$$d_{ij} = \max\{0,\, p_i - a_{ij} - p_j\}, \qquad (i, j) \in \mathcal{A}. \tag{3.3}$$

The scalars d_{ij} will be referred to as the *discrepancies* of the arcs (i, j). They quantify the error from optimality of the path P generated by the algorithm, as shown in the following proposition.

Proposition 3.2: Let the AWPC algorithm terminate with a path P, and let P' be any other path from s to t. Then we have

$$L_P + \sum_{(i,j)\in P} d_{ij} \leq L_{P'} + \sum_{(i,j)\in P'} d_{ij}, \tag{3.4}$$

where L_P and $L_{P'}$ are the total lengths of P and P', and d_{ij} are the arc discrepancies of Eq. (3.3), which are obtained upon termination.

Proof: Suppose that we increase the arc lengths a_{ij} by the corresponding arc discrepancies d_{ij} that are obtained upon termination, thus changing these lengths to

$$\bar{a}_{ij} = a_{ij} + d_{ij}, \qquad (i, j) \in \mathcal{A}.$$

Then upon termination, the path P produced by the AWPC algorithm is shortest with respect to arc lengths $\bar{a}_{ij}$. The reason is that the arcs that belong to P are level with respect to the arc lengths $\bar{a}_{ij}$, while the arcs that do not belong to P are either level or uphill, again with respect to $\bar{a}_{ij}$; this is the optimality condition

for P to be shortest with respect to $\bar{a}_{ij}$. The inequality (3.4) then follows, since its left and right sides are the lengths of P and P', respectively, with respect to $\bar{a}_{ij}$. **Q.E.D.**

The discrepancies d_{ij} provide an upper bound to the degree of suboptimality of the path obtained upon termination, as per Eq. (3.4). In particular, if $\overline{D}$ is the maximum arc discrepancy upon termination of the algorithm,

$$\overline{D} = \max_{(i,j)\in\mathcal{A}} d_{ij},$$

then, since $d_{ij} \geq 0$ for all (i,j), Eq. (3.4) implies that

$$L_P \leq L_{P'} + (n+1)\overline{D}, \tag{3.5}$$

where n is the number of nodes other than s and t. The reason is that a path from s to t can contain at most $(n+1)$ arcs, each having a discrepancy that is at most $\overline{D}$.

An interesting empirical observation is that when the algorithm creates a new downhill arc (i,j) that lies outside P, the corresponding discrepancy d_{ij} becomes equal to ϵ or a small multiple of ϵ. A reasonable conjecture is that if all the discrepancies d_{ij} are initially bounded by a small multiple of ϵ, then the path produced by the algorithm upon termination is shortest to within a small multiple of $n\epsilon$, where n is the number of nodes other than s and t. This bears similarity to auction algorithms for assignment and other network optimization problems.

3.4 ϵ-Complementary Slackness - Using ϵ-Scaling to Find a Shortest Path

The tradeoff between speed of convergence and accuracy of solution that is embodied in the choice of ϵ was recognized in the original proposal of the auction algorithm for the assignment problem [Ber79], and the approach of *ϵ-scaling* was proposed to deal with it. In this approach we start the auction algorithm with a relatively large value of ϵ, to obtain quickly rough estimates for appropriate values of the node prices, and then we progressively reduce ϵ to refine the node prices and eventually obtain an optimal solution. The use of ϵ-scaling also allows the option of stopping the algorithm, with a less refined solution, if the allotted time for computation is limited.

In the context of the AWPC algorithm, ϵ-scaling is implemented by running the algorithm with a relatively large value of ϵ to estimate "good" prices, at least for a subset of "promising" transit nodes from s to t, and then progressively refining the assessment of the "promise" of these nodes. This is done by rerunning the algorithm with smaller values of ϵ, while using as initial prices at each run the final prices of the previous run. It is well-known that ϵ-scaling improves the computational complexity of auction algorithms,† and it

† See the papers [BeE88], [Ber88], the book [Ber98], and the references quoted there, for polynomial complexity analyses of auction algorithms for the assignment problem and other related problems.

can be similarly applied to the AWPC algorithm, as we will discuss shortly.

Empirically, this scheme seems to work well, but it does not offer a guarantee that it will yield a shortest path. As an example in the problem of Fig. 3.4, if all prices are chosen to be 0 except for the price of node 2', which is chosen to be a high positive number, the AWPC algorithm will not find the shortest path for any value of ϵ.

We will now consider a variant of the AWPC algorithm and corresponding ϵ-scaling scheme that guarantee that a shortest path can be obtained. The ϵ-scaling scheme modifies the prices produced by the AWPC algorithm for a given value of ϵ, before applying the algorithm with a smaller value of ϵ. To this end, we will operate the AWPC algorithm so that it initially satisfies and subsequently maintains the following *ϵ-complementary slackness* condition (ϵ-CS for short).

ϵ-Complementary Slackness:

For a given $\epsilon > 0$, the prices $\{p_i \mid i \in \mathcal{N}\}$ and the path P satisfy

$$p_i \leq a_{ij} + p_j + \epsilon, \qquad \text{for all arcs } (i,j),$$

i.e., every arc is uphill, or level, or downhill by at most ϵ, and

$$p_i \geq a_{ij} + p_j, \qquad \text{for all arcs } (i,j) \text{ of the path } P,$$

i.e., every arc of P is level or downhill (by at most ϵ).

The notion of ϵ-CS is fundamental in the context of auction algorithms, and represents a relaxation of the classical complementary slackness condition of linear programming (see, e.g., Bertsimas and Tsitsiklis [BeT97]). In particular, when ϵ-CS holds, the discrepancies d_{ij} of Eq. (3.3) are at most equal to ϵ, so *if the AWPC algorithm maintains ϵ-CS throughout its operation, it produces a path that is suboptimal by at most* $(n+1)\epsilon$ in view of Eq. (3.5), and hence also optimal for ϵ sufficiently small [$(n+1)\epsilon$ should be less than the difference between the 2nd shortest path distance and the shortest path distance]. Thus maintaining ϵ-CS is desirable.

On the other hand, the AWPC algorithm need not maintain ϵ-CS throughout its operation, because the increase of p_{n_k} prior to an extension may violate the ϵ-CS inequality $p_{n_k} \leq a_{n_k j} + p_j + \epsilon$ for $j = \text{succ}(n_k)$ and possibly for j equal to some other downstream neighbors of n_k. A simple remedy is to choose the price increase prior to an extension in a specific way. In particular, in case (c1) of the AWPC algorithm, we raise the price p_{n_k} to the largest value that satisfies ϵ-CS, while extending P to $\text{succ}(n_k)$, rather than setting p_{n_k} to any value that makes the arc $\big(\text{pred}(n_k), n_k\big)$ level or downhill and the arc $\big(n_k, \text{succ}(n_k)\big)$ downhill.

More specifically, there are three cases to consider when the relation

$$p_{\text{pred}(n_k)} > a_{\text{pred}(n_k)n_k} + a_{n_k}\text{succ}(n_k) + p_{\text{succ}(n_k)}, \tag{3.6}$$

holds, which are illustrated in Fig. 3.5:

(a) **The arc $\big(\textbf{pred}(n_k), n_k\big)$ is level**, i.e.,

$$0 = p_{\text{pred}(n_k)} - a_{\text{pred}(n_k)n_k} - p_{n_k},$$

$$0 < p_{n_k} - a_{n_k}\text{succ}(n_k) - p_{\text{succ}(n_k)}.$$

Then we extend to $\text{succ}(n_k)$ and leave the price p_{n_k} unchanged (top part of Fig. 3.5).

(b) **The arc $\big(\textbf{pred}(n_k), n_k\big)$ is downhill and the arc $\big(n_k, \textbf{succ}(n_k)\big)$ is downhill or level**, i.e.,

$$0 < p_{\text{pred}(n_k)} - a_{\text{pred}(n_k)n_k} - p_{n_k},$$

$$0 \leq p_{n_k} - a_{n_k}\text{succ}(n_k) - p_{\text{succ}(n_k)}.$$

Then we extend to $\text{succ}(n_k)$ and set the price p_{n_k} to

$$\min\big\{p_{\text{pred}(n_k)} - a_{\text{pred}(n_k)n_k},\, a_{n_k}\text{succ}(n_k) + p_{\text{succ}(n_k)} + \epsilon\big\},$$

thus making the arc $\big(\text{pred}(n_k), n_k\big)$ is downhill or level and the arc $\big(n_k, \text{succ}(n_k)\big)$ downhill (middle part of Fig. 3.5).

(c) **The arc $\big(\textbf{pred}(n_k), n_k\big)$ is downhill and the arc $\big(n_k, \textbf{succ}(n_k)\big)$ is uphill**, i.e.,

$$0 < p_{\text{pred}(n_k)} - a_{\text{pred}(n_k)n_k} - p_{n_k},$$

$$p_{n_k} - a_{n_k}\text{succ}(n_k) - p_{\text{succ}(n_k)} < 0.$$

Then we extend to $\text{succ}(n_k)$ and set the price p_{n_k} to

$$p_{\text{pred}(n_k)} - a_{\text{pred}(n_k)n_k},$$

thus making the arc $\big(\text{pred}(n_k), n_k\big)$ level (if ϵ-CS is satisfied on this arc) or downhill (if ϵ-CS is violated on this arc), and the arc $\big(n_k, n_k\text{succ}(n_k)\big)$ downhill (bottom part of Fig. 3.5).

We refer to the AWPC algorithm with the specific price change rule given above as the *AWPC-CS algorithm*. It is a special case of the AWPC algorithm, and therefore it maintains the downhill path property, thus guaranteeing termination. It also ensures that the ϵ-CS property holds upon termination, provided the

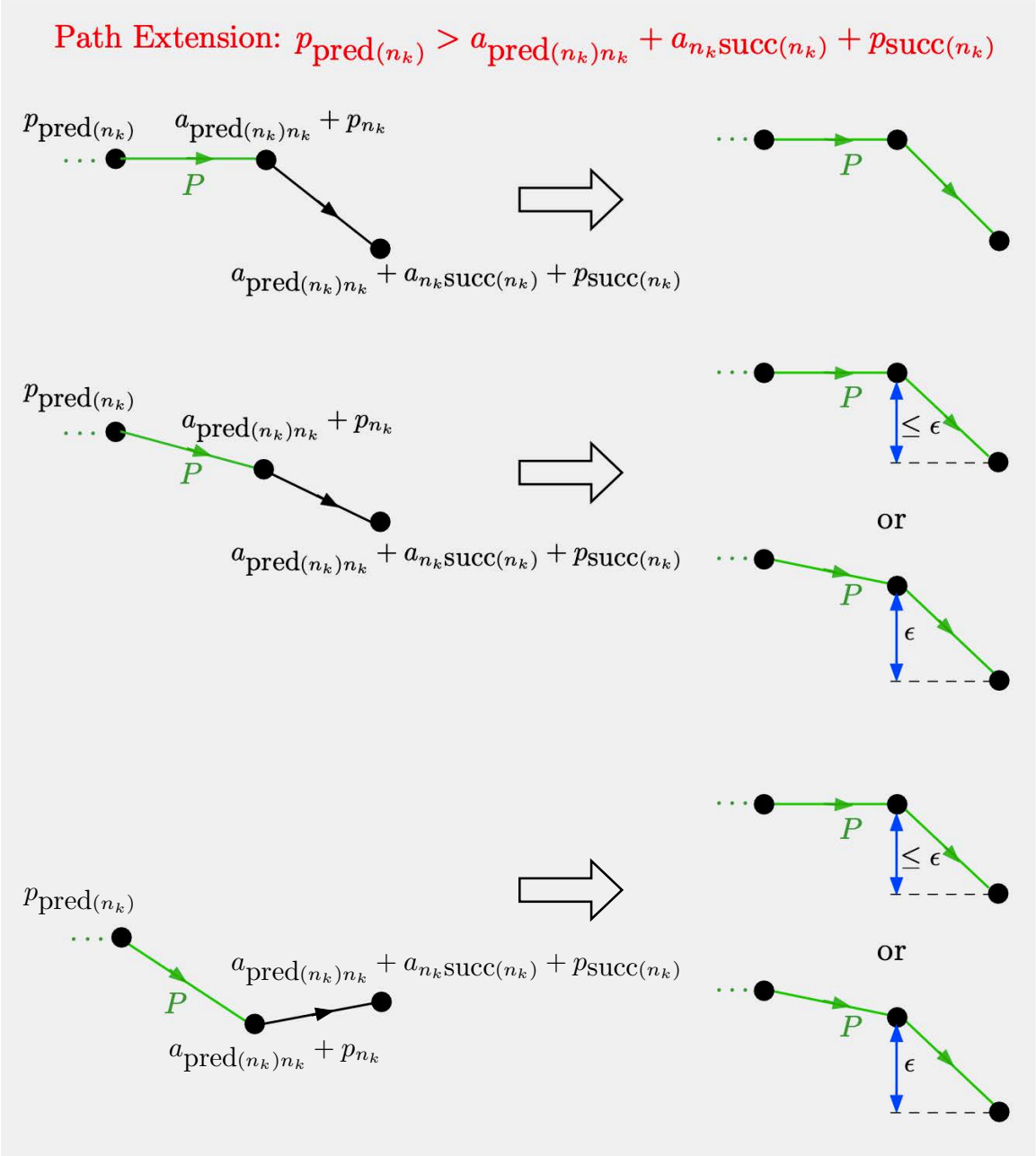


Figure 3.5 Illustration of an iteration of the AWPC-CS algorithm, which maintains ϵ-CS if started with prices satisfying ϵ-CS. The figure shows the levels

$$p_{\text{pred}(n_k)}, \qquad a_{\text{pred}(n_k)n_k} + p_{n_k}, \qquad a_{\text{pred}(n_k)n_k} + a_{n_k\text{succ}(n_k)} + p_{\text{succ}(n_k)},$$

before and after an extension. The middle and bottom portions of the figure illustrate how following the extension, it is possible that both arcs $\big(\text{pred}(n_k), n_k\big)$ and $\big(n_k, \text{succ}(n_k)\big)$ are downhill. In the case where ϵ-CS is satisfied on the arc $\big(\text{pred}(n_k), n_k\big)$, the arc will become level following the price rise.

initial price discrepancies are bounded by ϵ. As a result the path obtained upon termination is shortest to within $(n+1)\epsilon$; cf. Eq. (3.5).

We summarize the preceding arguments in the following proposition.

Proposition 3.3: Assume that there exists at least one path from the origin to the destination, and the nonnegative cycle condition (3.1) holds. Then the AWPC-CS algorithm terminates with a path from s to t. If in addition the initial prices satisfy ϵ-CS, then the algorithm will maintain the ϵ-CS property throughout its operation, and the path obtained termination will be shortest to within $(n+1)\epsilon$, where n is the number of nodes other than s and t. Otherwise the algorithm will never terminate and we have $p_i \to \infty$ for all nodes i in a subset $\mathcal{N}_\infty$ that contains s.

When the starting prices violate ϵ-CS for some arcs, it is possible that the ϵ-CS property is accidentally restored at some point during the operation of the AWPC-CS algorithm, in which case ϵ-CS will hold upon termination as per the preceding discussion. However, there is no guarantee that this will happen. On the other hand, as Fig. 3.5 illustrates, the AWPC-CS algorithm does not create new arcs that violate ϵ-CS. Thus *the algorithm has a tendency to self-correct*. In fact the following property of the algorithm can be verified: *the maximum arc violation of ϵ-CS, as measured by*

$$\overline{D}_\epsilon = \max_{(i,j)\in\mathcal{A}} \max\{0,\, p_i - a_{ij} - p_j - \epsilon\},$$

is not increased at any iteration.

Implementing ϵ-Scaling

Given the final set of prices and path obtained by the AWPC-CS algorithm for a given value of ϵ, there is an important issue in ϵ-scaling: how to modify the prices of some of the nodes so that the resulting prices together with the degenerate path (s) satisfy ϵ'-CS for a smaller positive value $\epsilon' < \epsilon$. Moreover the price modifications should be small in order for the new prices to be good starting points for rerunning the algorithm with the new value ϵ'.

There are algorithms for computing price modifications to satisfy ϵ'-CS for a smaller value $\epsilon' < \epsilon$ together with the degenerate path (s), which will be discussed in future reports. Moreover, often such algorithms can take advantage of special structure of the problem's graph. This is true for example in assignment problems, where the bipartite character of the graph allows great flexibility in the choice of the initial prices. In what follows in this section, we discuss the case of shortest path problems involving an acyclic graph, which arise prominently in on-line and off-line multistep lookahead minimization, and tree

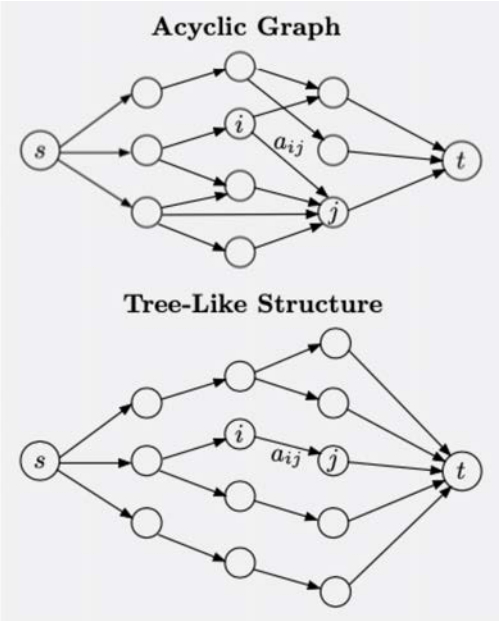


Figure 3.6 Illustration of an acyclic graph involving paths that start at s and end at t and arc lengths a_{ij}. A case of special interest in reinforcement learning is a tree-like structure, illustrated in the bottom figure, where nodes are grouped in layers, with arcs starting from one layer and ending at a node of the next layer, and there is a single incoming arc to each node except s and t.

search for reinforcement learning problems (see Section 6). A related algorithm for graphs with cycles is given in [Ber91], Section 4.2.

ϵ-Scaling in Acyclic Graphs

Let us consider the case of an acyclic graph; cf. Fig. 3.6. Suppose that we are given arc lengths a_{ij} and a set of prices $\{p_i \mid i \in \mathcal{N}\}$ that for a given positive ϵ satisfy

$$p_i \leq a_{ij} + p_j + \epsilon, \qquad \text{for all arcs } (i,j), \tag{3.7}$$

possibly resulting from application of the AWPC-CS algorithm to the corresponding shortest path problem. We want to find a set of prices $\{p'_i \mid i \in \mathcal{N}\}$ that for a given positive $\epsilon' < \epsilon$, satisfy

$$p'_i \leq a_{ij} + p'_j + \epsilon', \qquad \text{for all arcs } (i,j), \tag{3.8}$$

satisfy ϵ'-CS together with the degenerate path $P = (s)$. Thus Eq. (3.7) requires that arcs, when downhill, are downhill by at most ϵ, while Eq. (3.8) requires them to be downhill by at most ϵ'.

The idea is to start at t and sequentially proceed backwards towards s, by delineating arcs (i,j) that violate the condition (3.8) and raising the price of j, and possibly the prices of some descendants of j [since increasing p_j may violate ϵ'-CS for nodes that lie downstream of j]. Thus we must check descendants of j all

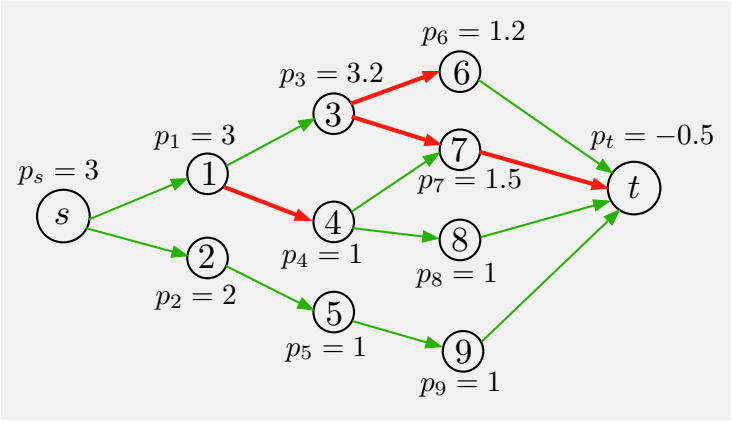


Figure 3.7 Illustration of ϵ-scaling in an acyclic graph, and the modifications needed to pass from the path $P = (s)$ and prices satisfying ϵ-CS to prices satisfying ϵ'-CS, with $\epsilon' < \epsilon$. All arcs have length equal to 1, and the prices p_i are shown next to the nodes i. Let $\epsilon = 1$ and $\epsilon' = 1/2$. All arcs satisfy the 1-CS condition (3.7), but arcs $(1, 4)$, $(3, 6)$, $(3, 7)$, and $(7, t)$ (shown in red) violate the 0.5-CS condition (3.8). We obtain prices satisfying 0.5-CS, by increasing the prices of the end nodes t, 6, 7, and 1 (in that order), and possibly their descendants in four iterations:

(1) Set $p_t = -0.5 \uparrow 0$.

(2) Set $p_6 = 1.2 \uparrow 1.7$ and $p_t = 0 \uparrow 0.2$.

(3) Set $p_7 = 1.5 \uparrow 1.7$ (no need to increase p_t further).

(4) Set $p_4 = 1 \uparrow 1.5$ (no need to increase p_7 or p_8, and hence also p_t).

the way to the destination t, and raise their prices by whatever amounts are necessary to enforce the ϵ'-CS condition (3.8) on arc (i, j). Figure 3.7 provides an example.

The idea of successively raising the prices of the end nodes j of arcs (i, j) that violate the condition (3.8), while keeping the price of the origin s unchanged, also works for nonacyclic graphs. After a finite number of price increases, the condition (3.8) will be satisfied for all arcs. However, the number of price increases required cannot be easily predicted in the absence of special structure.

Approximate ϵ-Scaling

While the AWPC-CS algorithm maintains the ϵ-CS property if this property is initially satisfied, finding initial prices that satisfy ϵ-CS may not be easy [except when $a_{ij} \geq 0$ for all arcs (i, j), in which case we can take $p_i = 0$ for all nodes i; algorithms for the more general case are given in [Ber91], Props. 6 and 7]. Moreover, operating ϵ-scaling is complicated when arc lengths change over time and on-line replanning is necessary. This motivates a heuristic scheme to select prices that satisfy the ϵ-CS inequality

$$p_i \leq a_{ij} + p_j + \epsilon,$$

for as many arcs (i,j) as is conveniently possible, and rely on the self-correcting mechanism of the algorithm, discussed earlier, to produce a high quality solution. A similar approach may be followed for the ϵ-scaling process: it may be performed approximately, in some heuristic and computationally inexpensive way.

A reasonable approach for enforcing the ϵ'-CS property selectively is to raise the prices of just the nodes on the final path P earlier constructed by the algorithm with a larger value $\epsilon > \epsilon'$. Here, we may start from s and go forward towards t along P, while raising the prices of the nodes of P, as necessary to enforce ϵ'-CS on the arcs of P (but not on arcs outside of P). A potential benefit of this idea in some contexts is that it provides an incentive for the algorithm to explore alternative paths to P.

3.5 A Variant with Optimistic Extensions

Let us now discuss a variant of the AWPC algorithm, which aims to accelerate convergence by performing an extension instead of a contraction in the special case where the current path $P = (s, n_1, \ldots, n_k)$ consists of multiple nodes [i.e., $P \neq (s)$], and we have

$$p_{\text{pred}(n_k)} = a_{\text{pred}(n_k)n_k} + a_{n_k\text{succ}(n_k)} + p_{\text{succ}(n_k)}. \tag{3.9}$$

According to case (c2) of the AWPC algorithm, we must then perform a contraction of P to $\text{pred}(n_k)$ and raise the price of n_k to

$$a_{n_k\text{succ}(n_k)} + p_{\text{succ}(n_k)} + \epsilon. \tag{3.10}$$

In the variant considered in this section, called *AWPC with optimistic extensions* (AWPC-OE for short), we consider two complementary cases:

(a) $\text{succ}(n_k) \notin P$, in which case we extend P to $\text{succ}(n_k)$, and we raise the price of n_k to

$$a_{n_k\text{succ}(n_k)} + p_{\text{succ}(n_k)}$$

[thus making both arcs $\big(\text{pred}(n_k), n_k\big)$ and $\big(n_k, \text{succ}(n_k)\big)$ level, while maintaining the acyclicity of P].

(b) $\text{succ}(n_k) \in P$, in which case we perform a contraction of P to $\text{pred}(n_k)$, and raise the price of $\text{succ}(n_k)$ to the level (3.10), as in the AWPC algorithm.

While in this variant, the acyclicity of P is maintained at each iteration, the downhill path property as stated in Section 3.1 does not hold anymore, because while each arc of P is either level or downhill, it is possible that all of them are level. Still, however, the convergence proof of Prop. 3.1 goes through and the algorithm is valid, the critical part being that the path P remains acyclic throughout the algorithm, so that an infinite number of node contractions must be performed if the algorithm does not terminate.

For an illustration, consider the problem of Example 3.1. In reference to Fig. 3.3, the AWPC-OE algorithm will generate the same iterations as the AWPC algorithm in the first three iterations to obtain

path $P = (s, 1, 2, 3)$, price vector $(p_s, p_1, p_2, p_3, p_t) = (\epsilon, \epsilon, \epsilon, 0, 0)$, $\text{pred}(3) = 2$, $\text{succ}(3) = 1$, and $p_{\text{pred}(3)} = p_{\text{succ}(3)} = \epsilon$. Then, Eq. (3.9) holds and the AWPC-OE algorithm will consider an extension to node 1, but since node 1 belongs to P, it will forego the extension, and perform a contraction to node 2, and continue exactly as the AWPC algorithm; cf. Fig. 3.3.

It is possible to refine the AWPC-OE algorithm for the case where Eq. (3.9) holds and there are multiple downstream neighbors j of n_k that have minimum value of $a_{n_k j} + p_j$. Then we can select one of these neighbors that does not belong to P and perform an extension, and perform a contraction if no such neighbor can be found. Thus if there are multiple neighbors that are candidates for $\text{succ}(n_k)$, we break the tie in favor of one that does not belong to P (if one exists) and perform an extension to that neighbor. This refinement may provide additional acceleration in special types of problems, such as unweighted path construction, where multiple neighbor candidates arise frequently.

In summary, the AWPC-OE algorithm allows an extension in some cases where the AWPC algorithm performs a contraction, and is likely to terminate faster. For this it must check to make sure that no cycle is closed through an extension in the case where Eq. (3.9) holds, since the downhill path property as stated in Section 3.1 may not hold.

4. ALGORITHMIC VARIANTS

In this section we outline variants of the APC and AWPC algorithms. Detailed development of some of these variants as well as modifications aimed at enhancing computational efficiency will be provided in future reports.

4.1 Multiple Destinations or Multiple Origins

The generalization of our algorithms to handle a single origin but multiple destinations is straightforward. We simply maintain a list of destinations that have not yet been reached by the path P, and we run the algorithm as if there was a single destination. Once another destination is reached by P, we remove this destination from the list. We then continue similarly, until all destinations are reached. A similar approach has been used to extend the auction/shortest path algorithm of [Ber91] to the case of multiple destinations.

It is also possible to generalize our algorithms to handle multiple origins but a single destination. We simply run the algorithms one origin at a time, as if there was a single origin, and continue similarly, until a path has been constructed starting from every origin. With multiple origins, however, there is a time-saving possibility. Suppose that we have constructed a path P_1 starting at origin 1 and ending at t, and then while constructing a path P_2 that starts at origin 2, we land upon a node $n_1 \neq t$ of P_1 through an extension. Then we can simply join P_2 with the tail portion of P_1 that starts at n_1 and construct a complete path that starts

at node 2 and ends at t. This can be repeated for all origins, thus eventually constructing a tree of paths to the destination.

4.2 Forward/Reverse Path Construction Algorithms

This variant is inspired by forward/reverse versions of the auction algorithm for the assignment problem, due to Bertsekas, Castañon, and Tsaknakis [BCT93], which have also been described and extended in the book [Ber98]. When applied to the shortest path context, this idea involves maintaining a forward path that starts from the origin, as well as a reverse path that ends at the destination. The forward path construction uses price increases and proceeds from the origin towards the destination, while the reverse path uses price reductions and proceeds backwards from the destination towards the origin. The forward and the reverse algorithms are symmetric replicas of each other. For the case of a single destination, the algorithm terminates when the forward path that starts from the origin meets the backward path that starts from the destination. A forward/reverse auction algorithm for shortest paths is given in [Ber91], Section 4.1.

It is well established by computational practice that forward/reverse variants of auction algorithms generally work faster (and often much faster) than the forward or the reverse algorithms operating alone. It is expected that forward/reverse variants of our algorithms will similarly work much more efficiently than just corresponding forward algorithms. Thus forward/reverse versions of our algorithms are an important research direction to pursue.

4.3 Distributed Implementations

Auction algorithms are well-suited for parallelization, as extensive implementation studies for the assignment and other network flow problems have shown (see e.g., the book by Bertsekas and Tsitsiklis [BeT89], and the papers by Bertsekas and Castañon [BeC91], Bertsekas et al. [BCE95], Beraldi, Guerriero, and Musmanno [BGM97], Zavlanos, Spesivtsev, and Pappas [ZSP08], and Naparstek and Leshem [NaL16]). Possibilities for parallel and distributed asynchronous computation similarly arise within our context when there are multiple destinations and/or multiple origins. The idea is to construct multiple paths simultaneously, with shared price use and asynchronous price updating during the path construction process. Such possibilities are an interesting direction for further research, and have been explored for a related type of auction/shortest path algorithm by Polymenakos and Bertsekas [PoB94].

4.4 Algorithms Based on Transformations to Equivalent Matching or Assignment Problems

Path construction problems, weighted and unweighted, can be converted to equivalent assignment and unweighted matching problems, respectively, by using well-known transformations (see e.g., Fig. 1.1, and the

book [Ber98], which also describes several types of other transformations of network optimization problems to assignment problems). Using these transformations, we can apply the auction algorithm for assignment, which has been investigated extensively. The resulting algorithms bear considerable resemblance to our algorithms. A comparative evaluation of these alternative algorithms and their specialized variations is an interesting subject for further research.

4.5 Variants Involving Multiple Node Price Rises

We have discussed so far algorithms that involve a price rise at just the terminal node of the path P maintained by the algorithm. However, a simultaneous contraction and attendant price rise at multiple nodes of P may be possible. Of course this must be done in a way to preserve the downhill path property in some modified form, whereby all arcs of P are either level or downhill, with the last arc of P being downhill following an extension. Such simultaneous price rises may be beneficial if they economize in subsequent computation, and can be facilitated by suitable implementation. For example, when extending P from n_k to n_{k+1}, we may store the "second best neighbor" n'_{k+1} and corresponding "second best value"

$$a_{n_k n'_{k+1}} + p_{n'_{k+1}},$$

where n'_{k+1} is the node that minimizes $a_{n_k n} + p_n$ over all $n \neq n_{k+1}$ such that (n_k, n) is an arc. This information can be used at future iterations to determine efficiently if multiple contractions along P can be performed simultaneously. This and other related implementation ideas have been discussed in the papers [Ber95a] and [Ber95b], and have been incorporated in efficient max-flow and minimum cost flow codes, which are available from the author's web site.

5. PATH-BASED AUCTION ALGORITHMS FOR NETWORK TRANSPORT

In this section we will consider a general single-commodity network optimization problem and a broad extension of our auction/path construction approach for solving it. The problem involves a network with a single source, a single sink, and a given amount of supply to be transported from the source to the sink, while respecting given arc capacities. We will use our path construction algorithms as the basis for a methodology to solve (exactly) the unweighted version of this problem, and (inexactly) the weighted version of the problem. Our methods involve successive path constructions, flow augmentations along the constructed paths, and reuse of prices from one path construction to the next. The methods also involve a positive ϵ parameter, whose choice embodies a tradeoff between accuracy of solution and speed of convergence, and allows for the use of ϵ-scaling.

5.1 The Unweighted Version of the Problem

In the unweighted version of the problem we want to transfer a given amount of flow from a source node to a sink node in a given network without regard for the cost of the transfer. In particular, we have a directed graph with set of nodes $\mathcal{N}$ and set of arcs $\mathcal{A}$. There are two special nodes, denoted s and t, which are called the *source* and *sink*, respectively. We assume that there are no incoming arcs to the source and no outgoing arcs from the sink. Each arc (i,j) is to carry a flow x_{ij} that must satisfy a constraint of the form

$$0 \leq x_{ij} \leq c_{ij}, \qquad \forall\ (i,j) \in \mathcal{A}.$$

Here c_{ij} is either a positive scalar or is equal to ∞. It represents the "capacity" of arc (i,j).

We are also given a positive scalar r, representing supply to be transported from s to t, and we consider the problem of finding a flow vector $\{x_{ij} \mid (i,j) \in \mathcal{A}\}$ that satisfies

$$\sum_{\{j \mid (i,j) \in \mathcal{A}\}} x_{ij} - \sum_{\{j \mid (j,i) \in \mathcal{A}\}} x_{ji} = 0, \qquad \forall\ i \in \mathcal{N},\ i \neq s,\ t, \tag{5.1}$$

$$\sum_{\{j \mid (s,j) \in \mathcal{A}\}} x_{sj} = \sum_{\{j \mid (j,t) \in \mathcal{A}\}} x_{jt} = r, \tag{5.2}$$

$$0 \leq x_{ij} \leq c_{ij}, \qquad \forall\ (i,j) \in \mathcal{A}. \tag{5.3}$$

This is a feasibility problem, whereby we want to transfer a given amount r of flow from the source node s to the sink node t, while satisfying the arc capacity constraints.† The problem can be solved with a classical approach, which is based on the idea of successive flow augmentations that start at the source, end at the sink and use paths within the so called reduced graph.‡ To this end we can use the APC algorithm of Section 2.1 to construct the paths for the successive flow augmentations, with reuse of the prices from one flow augmentation to next (i.e., use the final prices from one flow augmentation as the starting prices of the next flow augmentation). A similar auction algorithm with price reuse has been given for max-flow problems in [Ber95a], and tested extensively on large-scale instances with excellent computational results, and far superior performance over competing codes at the time (a code implementing this algorithm is available from the author's website).

Let us provide an example of this type of algorithm and an illustration of the reduced graph for the case of a matching problem.

---

† Sometimes this problem is called the *fixed flow problem*, to contrast it to the closely related max-flow problem, where we want to maximize the supply r, while maintaining feasibility with respect to arc capacities.

‡ The reduced graph is obtained from the original graph by deleting all arcs (i,j) for which $x_{ij} = c_{ij}$, and by introducing a new (reversed) arc (j,i) for each arc (i,j) such that $0 < x_{ij}$. An example of the reduced graph is provided in the subsequent Example 5.1 and Fig. 5.1. The reduced graph plays a central role in many network flow contexts, including max-flow, primal-dual, and auction algorithms; see [Ber98], starting with Section 3.3.

Example 5.1 (Successive Path Constructions for Solving a Matching Problem)

Let us consider an unweighted matching problem, where we want to assign three persons, denoted 1, 2, 3 to three objects denoted $1'$, $2'$, $3'$. We can transform the problem to a feasibility problem of the form (5.1)-(5.3), as shown in the top part of Fig. 5.1, with all arcs capacities equal to 1.

The first iteration of the APC algorithm will find some path from s to t in the original graph, and let's assume for the sake of illustration, that this path is $(s, 1, 2', t)$, matching person 1 to object $2'$. The reduced graph is then created by reversing the direction of arcs $(s, 1)$, $(2', t)$, and $(1, 2')$. The second iteration will find some path from s to t in this reduced graph. Let's assume, for the sake of illustration, that this path is $(s, 2, 3', t)$, matching person 2 to object $3'$. The resulting reduced graph is the one shown in the bottom part of Fig. 5.1. The third iteration will find one of the three possible paths from s to t in this reduced graph, resulting in one of the three matchings indicated in Fig. 5.1. The algorithm then terminates.

Generally, for $n \times n$ matching problems, our algorithm will consist of n path constructions, the first $(n-1)$ of which are followed by a suitable modification of the reduced graph. The final prices from each path construction are used as initial prices for the next path construction.

5.2 The Weighted Version of the Problem

In the linearly weighted version of the problem we have a cost a_{ij} per unit flow on each arc (i, j), and we want to transfer a given amount r of flow from s to t, while minimizing the total cost of the transfer. Thus we want to minimize

$$\sum_{(i,j)\in\mathcal{A}} a_{ij}x_{ij},$$

subject to the constraints (5.1)-(5.3). This is a classical problem of network transport, which contains as special cases problems of assignment (i.e., weighted matching), constrained shortest path (such as k node-disjoint paths), transportation, transhipment, etc, for which specialized auction algorithms have been developed (see [Ber88], [BeC89], [BeC93], and the survey [Ber92]). A well-known primal-dual algorithm to solve the problem is based on successive flow augmentations that start at the source, end at the sink, and use shortest paths within the reduced graph (see [Ber98], Chapter 6).

To this end we can use the AWPC algorithm to construct near-shortest paths for the successive flow augmentations, with reuse of the prices from one flow augmentation to the next. The resulting algorithm will find an approximately optimal solution, with the degree of suboptimality determined by the parameter ϵ. Like AWPC, this algorithm is new, but resembles other already existing auction algorithms. It has its roots in the auction sequential shortest path algorithms described in the author's paper [Ber95b] (which includes extensive computational comparisons with alternative auction and primal-dual algorithms) and in the book [Ber98], Chapter 7. In particular, the paper [Ber95b] describes several implementation variants and ideas

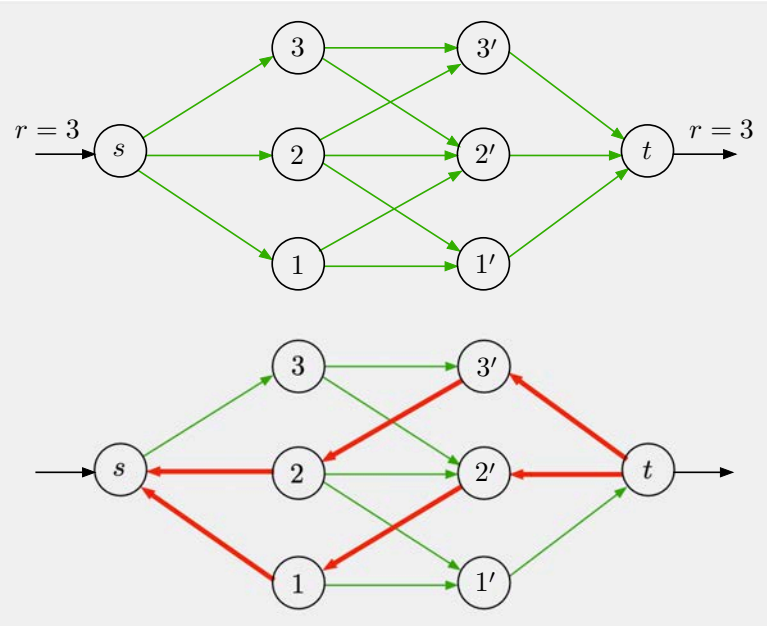


Figure 5.1 Illustration of the conversion of a 3×3 matching problem to a feasibility problem of the form (5.1)-(5.3) (see the top figure and Example 5.1). All arc capacities are equal to 1 and the supply r is equal to 3. In the bottom part of the figure, we show the reduced graph for the graph shown at the top, after node 1 has been assigned to node 2' and node 2 has been assigned to node 3'. We reverse the direction of the assigned arcs $(1, 2')$ and $(2, 3')$, and also of the corresponding arcs $(s, 1)$, $(s, 2)$, $(2', t)$, $(3', t)$. To complete the solution, we need to find a path from s to t in this reduced graph. For any set of initial prices, the APC algorithm will construct one of the following paths:

$(s, 3, 3', 2, 1', t)$ resulting in the matching $(1, 2'), (2, 1'), (3, 3')$,

$(s, 3, 3', 2, 2', 1, 1', t)$ resulting in the matching $(1, 1'), (2, 2', (3, 3')$,

$(s, 3, 2', 1, 1', t)$ resulting in the matching $(1, 1'), (2, 3'), (3, 2')$.

(saving path fragments, early flow augmentations, optimistic extensions, etc). These variants are applicable to the context of the present section and have been incorporated in efficient max-flow and minimum cost flow codes, which are available from the author's web site.

In the case of an assignment problem with a graph such as the one illustrated in Fig. 3.3 and suitable arc costs a_{ij}, the AWPC algorithm will operate similar to the APC algorithm in Example 5.1. For the 3×3 problem of Fig. 3.3, there will be three augmenting path constructions, guided by the prices and the arc costs. The first two of these will be followed by a suitable modification of the reduced graph. The final prices from each path construction will be used as initial prices for the next path construction. The price changes involved in the path construction are similar to the ones resulting from the bidding process of the original auction algorithm for the assignment problem [Ber79].

5.3 Convex Separable Cost Extensions

We finally note that auction algorithms for the linearly weighted version of the problem have been extended to network problems with convex separable cost functions of the form

$$\sum_{(i,j)\in\mathcal{A}} f_{ij}(x_{ij}),$$

where f_{ij} is a scalar convex function for each arc $(i,j) \in \mathcal{A}$. Here we want to find a flow vector $\{x_{ij} \mid (i,j) \in \mathcal{A}\}$ that minimizes this cost function subject to the constraints (5.1)-(5.3). We refer to the book [Ber98], Section 9.6 for descriptions, analysis, and references relating to auction algorithms with convex separable cost functions. These algorithms are based on generalized notions of ϵ-complementary slackness and ϵ-scaling, and can be adapted to use path constructions and corresponding flow augmentations, based on the ideas of this paper.

Note that a convex cost function may arise naturally for some problems, or it may be obtained from a linear cost function through some form of regularization. For examples of such applications, we refer to sources on matrix balancing and related problems (see e.g., the book by Censor and Zenios [CeZ97]), and sources on problems of network transport relating to approximate solution of the Monge-Ampere equation (see e.g., the books [Gal16], [PeC19], [San15], [Vil09], [Vil21] noted earlier).

A further generalized algorithm based on convex separable cost auction ideas addresses flow optimization in networks with gains, and has been given in the paper by Tseng and Bertsekas [TsB00]. This algorithm can also be adapted to use the methodology of the present paper.

6. CONNECTIONS WITH REINFORCEMENT LEARNING METHODS

Reinforcement learning (RL for short) is a popular approximation methodology for a large variety of sequential decision and control problems that can in principle be dealt with by dynamic programming (DP for short). It is well-known that every finite-state deterministic DP problem can be posed as a shortest path problem over an acyclic graph, with the origin node corresponding to the initial state of the DP problem. It is therefore natural to expect that the path construction algorithms of the present paper can find substantial application within the context of RL. In this section, we will outline some of the connections of our methods with the broad RL methodology of *approximation in value space*, which is based on replacing the optimal cost function in the DP algorithm by an approximation.

Approximation in value space with one-step or multistep lookahead minimization lies at the heart of many prominent artificial intelligence successes, including AlphaZero and other related game programs. It is also representative of the methods of rolling and receding horizon control, including model predictive

control, which have been used with success for many years in control system design and operations research applications. Generally, in such problems we have a dynamic system that generates a sequence of transitions between states under the influence of decision/control over a finite or infinite number of steps, and with a cost for every transition. The objective is to select the decisions to minimize the sum of all the transition costs. For example in the s-to-t shortest path problem, the states are the nodes, with s and t being the initial and final states, respectively, the decision/control at a node is the choice of a downstream neighbor node, and the transition cost is the length of the corresponding arc.

A useful viewpoint, which has been emphasized in the author's recent book [Ber22b], is to think of approximation in value space schemes as consisting of two components:

(a) *The off-line training algorithm*, which "learns" a value function and possibly a default policy by using data, either externally given or self-generated by simulation. The value function provides an estimate of the cost of starting at any one state, while the default policy supplies a (suboptimal) decision/control at any one state.

(b) *The on-line play algorithm*, which generates decisions in real time using the value function and possibly the policy that has been obtained by off-line training. This algorithm is invoked to select a decision at any state of the DP problem, once this state is generated in real time.

It is argued in the book [Ber22b] that the on-line play algorithm amounts to a step of Newton's method for solving Bellman's equation, while the starting point for the Newton step is determined by the results of off-line training. This supports a conceptual idea that applies in great generality and is central in the book, namely that *the performance of an off-line trained policy can be greatly improved by on-line play*.

We next discuss how our path construction algorithms of this paper can be blended into approximation in value space schemes for deterministic DP problems, as well as into some schemes that apply to stochastic DP problems.

Path Construction in the Context of Off-Line Training

The analysis of Section 3.3 suggests that the initial prices p_i in the AWPC algorithm should be chosen to be close to the shortest distances D_i^*, or more accurately, they should be chosen in a way that keeps the arcs nearly level or uphill, and minimizes the arc discrepancies; cf. Prop. 3.2. Of course we do not know the exact values D_i^*, but in a given application we may be able to use as initial prices approximate values, which can be obtained off-line through a computationally inexpensive heuristic or other machine learning methods. Collectively, these approximate values constitute a value function obtained by off-line training, to be used subsequently by the on-line play algorithm that is based on AWPC.

In one possible approach we may use data to train a neural network or other approximation architecture

to learn approximations to the shortest distances D_i^*. The data may be obtained by using a shortest path algorithm and arc lengths that are similar to the ones of the given problem. The training should also aim to produce prices for which the discrepancies d_{ij} given by Eq. (3.3) are small. This objective can and should be encoded into the training problem. It is also possible to train multiple neural networks to use for different patterns of arc lengths.

Path Construction in the Context of On-Line Play - Connections to Monte Carlo Tree Search

There are also possibilities for using the AWPC algorithm during on-line play, since several RL methods rely on the computation of (nearly) shortest paths on-line. In particular, on-line play schemes often use a *multistep lookahead search* for path construction through an acyclic decision graph. The search involves a graph traversal algorithm to reach the leaves of the graph, starting from the root node, which corresponds to the current state of the DP problem being solved on-line. It also uses a terminal cost at the leaves of the graph, which is obtained by using an off-line trained value function or by using a base heuristic on-line. The graph traversal may be done by using (nearly) shortest path calculations (see RL books such as the author's [Ber19], [Ber20a], [Ber22b], as well as the books by Sutton and Barto [SuB18], and Lattimore and Szepesvari [LaS20]). The techniques of real-time dynamic programming, described in the papers by Korf [Kor90], and Barto, Bradtke, and Singh [BBS95], among many others, are relevant in this context.

A popular class of methods for on-line play with multistep lookahead is *Monte Carlo tree search* (MCTS for short). These methods evaluate approximately the leaves of a tree of state transitions with root at the current state, combine the results of the evaluations by backwards propagation to the root of the tree, and progressively expand the depth of the tree by adding new leaves. A standard way to describe MCTS (see the surveys [BPW12] and [SGS21], and the book [LaS20]) is in terms of four components: *selection, expansion, simulation, and backup*. Selection refers to choosing a leaf node of the tree, to improve its evaluation, and possibly to add its descendants to the tree. Often in MCTS the selection is done by various criteria that try to balance exploration and exploitation, such as the statistics-based UCB (upper confidence bound) criterion. Expansion refers to the method used for tree enlargement, and may be based on the UCB criterion or other more traditional iterative deepening techniques (searching to adequate precision at a given level of lookahead before starting to search at a deeper level of lookahead). Simulation refers to the approximate evaluation of a leaf node by one or more stochastic Monte Carlo simulation runs. Finally, backup refers to the backwards propagation of the results of the leaf node evaluations to the root of the tree. The decision to be applied at the current state is the one corresponding to the best backed up evaluation. Note that while MCTS inherently assumes a stochastic decision environment, it has been applied to deterministic problems as well by using problem-dependent heuristics for tree pruning and expansion. Moreover, MCTS algorithms has been developed for adversarial contexts and games (even deterministic such as chess), in conjunction with techniques of minimax search such as alpha-beta pruning and others.

The problem that is solved approximately by MCTS is a shortest path problem with the origin being the root of the tree (the current state of the DP problem), and the destination being an artificial node to which all leaf nodes are connected with arcs that have the leaf evaluations as lengths. Thus for deterministic problems, one may consider the use of the AWPC algorithm as an alternative to MCTS for solving this shortest path problem. In particular, the selection and backup processes are replaced by the extension/contraction mechanism of the AWPC algorithm, while the simulation process may be performed by a deterministic base heuristic. The interim leaf evaluation results may be used for tree expansion in some more or less heuristic way. Each tree expansion may be followed by suitable price modifications to enforce an ϵ-CS condition, similar to the scheme discussed in Section 3.4 for acyclic graphs. One may also use ϵ-scaling at appropriate points to refine the quality of the solution. At some point the tree search is terminated, possibly upon reaching the limit of the computational budget. The decision at the current state is chosen to be the first arc of the final path generated by the AWPC algorithm. The analysis and implementation of the AWPC algorithm within search contexts where MCTS has traditionally been applied is an interesting subject for further research.

Rollout

An important algorithmic idea within the on-line play context is *rollout*. This is a popular class of RL methods that has received a lot of attention as an effective and easily implementable (suboptimal) methodology; see the books [Ber19], [Ber20a]. In a rollout algorithm, at each encountered state, we minimize over the decisions of the current stage, and treat the future stages approximately, through a relatively fast heuristic, called the *base heuristic*. An auction algorithm, including AWPC, could be a suitable base heuristic. As an example, the paper [Ber20b] illustrates applications of a combined auction/rollout algorithm for solution of multidimensional assignment problems.

Generally, in a rollout algorithm the idea is to use as value function the cost function of the base heuristic. The key property is that the performance of the rollout algorithm improves on the performance of the base heuristic. This is in the spirit of the fundamental DP method of policy iteration, which is intimately connected to rollout (the book [Ber20a] has a special focus on rollout and related methods that also apply to multiagent problems).

Shortest Path Construction for On-Line Play in Stochastic and Time-Varying Environments

The AWPC algorithm is inherently deterministic, but it can also be applied in stochastic multistep lookahead contexts, where the Monte Carlo tree search methods have been used widely. This can be done by replacing all steps of a multistep lookahead *except for the first* by deterministic approximations through the use of certainty equivalence (replacing stochastic quantities by fixed deterministic substitutes; see e.g., the book

[Ber19]). The deterministic shortest path optimizations following the first step of lookahead involve an acyclic graph, and can be handled with the AWPC algorithm. There is good reason for taking into account the stochastic nature of the first step without approximation, in order to maintain the connection of the lookahead minimization with Newton's method for solving the Bellman equation, as has been explained in the book [Ber22b], Section 3.2.

Let us also mention a possibility that arises in a time-varying environment where some of the arc lengths may be changing, possibly because some arcs may become unavailable and new arcs may become available, while new instances of shortest path problems arise. This is also typical in problems of adaptive control. An interesting possibility may then be to update the initial prices using machine learning methodology, and a combination of off-line and on-line training with data.

In this regard, we should mention that the use of reinforcement learning (RL) methods in conjunction with our path construction algorithms is facilitated by the fact that the initial prices are unrestricted. This makes our algorithms well-suited for large-scale and time-varying environments, such as data mining and transportation, where requests for solution of path construction problems arise continuously over time. Addressing the special implementation and machine learning issues in the context of such environments is an interesting subject for further research.

In conclusion, there are several potentially fruitful possibilities to mesh the AWPC algorithm within the RL methodology. The key property is that these algorithms will produce a feasible path starting with arbitrary prices. This path will be near optimal if the starting prices are close to the true (unknown) shortest distances or if they satisfy an ϵ-CS condition with ϵ relatively small. Moreover, it is plausible that better paths can be obtained by more closely approximating the shortest distances using heuristics and training with data. This conjecture is supported by experience with related auction algorithms, but remains to be established empirically.

7. CONCLUDING REMARKS

In this paper we have introduced an alternative framework for the development of auction algorithms, namely path construction rather than the assignment problem. The new framework allows for arbitrary initial prices, unconstrained by complementary slackness conditions, and is thus well suited for approximations based on training with data, and on-line path replanning. The new framework may also be well suited for some classical application contexts such as shortest path, max-flow, transportation, and transhipment.

Much work remains to be done to explore the possibilities for application of our new auction approach within the broad framework of path planning and network transport. It is also interesting to explore potential applications within reinforcement learning contexts. Moreover, it will be important to delineate the

application areas where the new auction algorithms have superior computational performance over the existing ones, consistent with the empirical observations for max-flow problems [Ber95a] and minimum cost flow problems [Ber95b]. These objectives are part of the scope of the forthcoming monograph by the author [Ber22a].

8. REFERENCES

[BBS95] Barto, A. G., Bradtke, S. J., and Singh, S. P., 1995. "Learning to Act Using Real-Time Dynamic Programming," Artificial Intelligence, Vol. 72, pp. 81-138.

[BCE95] Bertsekas, D. P., Castañon, D. A., Eckstein, J., and Zenios, S., 1995. "Parallel Computing in Network Optimization," Handbooks in OR and MS, Ball, M. O., Magnanti, T. L., Monma, C. L., and Nemhauser, G. L. (eds.), Vol. 7, North-Holland, Amsterdam, pp. 331-399.

[BCT93] Bertsekas, D. P., Castañon, D. A., and Tsaknakis, H., 1993. "Reverse Auction and the Solution of Inequality Constrained Assignment Problems," SIAM J. on Optimization, Vol. 3, pp. 268-299.

[BFH03] Brenier, Y., Frisch, U., Henon, M., Loeper, G., Matarrese, S., Mohayaee, R., and Sobolevskii, A., 2003. "Reconstruction of the Early Universe as a Convex Optimization Problem," Monthly Notices of the Royal Astronomical Society, Vol. 346, pp. 501-524.

[BGM97] Beraldi, P., Guerriero, F., and Musmanno, R., 1997. "Efficient Parallel Algorithms for the Minimum Cost Flow Problem," Journal of Optimization Theory and Applications, Vol. 95, pp. 501-530.

[BPS95] Bertsekas, D. P., Pallottino, S., and Scutellà, M. G., 1995. "Polynomial Auction Algorithms for Shortest Paths," Computational Optimization and Applications, Vol. 4, pp. 99-125.

[BPW12] Browne, C., Powley, E., Whitehouse, D., Lucas, L., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., and Colton, S., 2012. "A Survey of Monte Carlo Tree Search Methods," IEEE Trans. on Computational Intelligence and AI in Games, Vol. 4, pp. 1-43.

[BSS08] Bayati, M., Shah, D., and Sharma, M., 2008. "Max-Product for Maximum Weight Matching: Convergence, Correctness, and LP Duality," IEEE Trans. on Information Theory, Vol. 54, pp. 1241-1251.

[BeC89] Bertsekas, D. P., and Castañon, D. A., 1989. "The Auction Algorithm for the Transportation Problem," Annals of Operations Research, Vol. 20, pp. 67-96.

[BeC91] Bertsekas, D. P., and Castañon, D. A., 1991. "Parallel Synchronous and Asynchronous Implementations of the Auction Algorithm," Parallel Computing, Vol. 17, pp. 707-732.

[BeC93] Bertsekas, D. P., and Castañon, D. A., 1993. "A Generic Auction Algorithm for the Minimum Cost Network Flow Problem," Computational Optimization and Applications, Vol. 2, pp. 229-260.

[BeE88] Bertsekas, D. P., and Eckstein, J., 1988. "Dual Coordinate Step Methods for Linear Network Flow Problems," Math. Programming, Series B, Vol. 42, pp. 203-243.

[BeT89] Bertsekas, D. P., and Tsitsiklis, J. N., 1989. Parallel and Distributed Computation: Numerical Methods, Prentice-Hall, Engl. Cliffs, N. J. (can be downloaded from the author's website).

[BeT97] Bertsimas, D., and Tsitsiklis, J. N., 1997. Introduction to Linear Optimization, Athena Scientific, Belmont, MA.

[Ber79] Bertsekas, D. P., 1979. "A Distributed Algorithm for the Assignment Problem," Lab. for Information and Decision Systems Report, MIT, May 1979.

[Ber88] Bertsekas, D. P., 1988. "The Auction Algorithm: A Distributed Relaxation Method for the Assignment Problem," Annals of Operations Research, Vol. 14, pp. 105-123.

[Ber91] Bertsekas, D. P., 1991. "An Auction Algorithm for Shortest Paths," SIAM J. on Optimization, Vol. 1, pp. 425-447.

[Ber92] Bertsekas, D. P., 1992. "Auction Algorithms for Network Flow Problems: A Tutorial Introduction," Computational Optimization and Applications, Vol. 1, pp. 7-66.

[Ber95a] Bertsekas, D. P., 1995. "An Auction Algorithm for the Max-Flow Problem," J. of Optimization Theory and Applications, Vol. 87, pp. 69-101.

[Ber95b] Bertsekas, D. P., 1995. "An Auction/Sequential Shortest Path Algorithm for the Minimum Cost Network Flow Problem," Report LIDS-P-2146, MIT.

[Ber98] Bertsekas, D. P., 1998. Network Optimization: Continuous and Discrete Models, Athena Scientific, Belmont, MA (also available on-line from the author's website).

[Ber17] Bertsekas, D. P., 2017. Dynamic Programming and Optimal Control, Vol. I, Athena Scientific, Belmont, MA.

[Ber19] Bertsekas, D. P., 2019. Reinforcement Learning and Optimal Control, Athena Scientific, Belmont, MA.

[Ber20a] Bertsekas, D. P., 2020. Rollout, Policy Iteration, and Distributed Reinforcement Learning, Athena Scientific, Belmont, MA.

[Ber20b] Bertsekas, D. P., 2020. "Constrained Multiagent Rollout and Multidimensional Assignment with

the Auction Algorithm," arXiv preprint, arXiv:2002.07407.

[Ber22a] Bertsekas, D. P., 2022. Auction Algorithms for Assignment, Path Planning, and Network Transport, Athena Scientific, Belmont, MA (in preparation).

[Ber22b] Bertsekas, D. P., 2022. Lessons from AlphaZero for Optimal, Model Predictive, and Adaptive Control, Athena Scientific, Belmont, MA (also available as an ebook from Google Books, and on-line from the author's website).

[BiT22] Bicciato, A., and Torsello, A., 2022. "GAMS: Graph Augmentation with Module Swapping," Proc. of ICPRAM, pp. 249-255.

[CLG22] Clark, A., de Las Casas, D., Guy, A., Mensch, A., Paganini, M., Hoffmann, J., Damoc, B., Hechtman, B., Cai, T., Borgeaud, S,. and Van Den Driessche, G. B., 2022. "Unified Scaling Laws for Routed Language Models," Proc. International Conference on Machine Learning, pp. 4057-4086.

[CeZ97] Censor, Y., and Zenios, S. A., 1997. 'Parallel Optimization: Theory, Algorithms, and Applications, Oxford University Press.

[Gal16] Galichon, A., 2016. Optimal Transport Methods in Economics, Princeton University Press.

[JME18] Jacobs, M., Merkurjev, E. and Esedoglu, S., 2018. "Auction Dynamics: A Volume Constrained MBO Scheme," Journal of Computational Physics, Vol. 354, pp. 288-310.

[KoY94] Kosowsky, J. J., and Yuille, A. L., 1994. "The Invisible Hand Algorithm: Solving the Assignment Problem with Statistical Physics," Neural Networks, Vol. 7, pp. 477-490.

[Kor90] Korf, R. E., 1990. "Real-Time Heuristic Search," Artificial Intelligence, Vol. 42, pp. 189-211.

[LBD21] Lewis, M., Bhosale, S., Dettmers, T., Goyal, N., and Zettlemoyer, L., 2021. "Base Layers: Simplifying Training of Large, Sparse Models," Proc. International Conference on Machine Learning, pp. 6265-6274.

[LaS20] Lattimore, T., and Szepesvari, C., 2020. Bandit Algorithms, Cambridge University Press.

[MeT21] Merigot, Q., and Thibert, B., 2021. "Optimal Transport: Discretization and Algorithms," in Handbook of Numerical Analysis, Elsevier, Vol. 22, pp. 133-212.

[NaL16] Naparstek, O., and Leshem, A., 2016. "Expected Time Complexity of the Auction Algorithm and the Push Relabel Algorithm for Maximum Bipartite Matching on Random Graphs," Random Structures and Algorithms, Vol. 48, pp. 384-395.

[PeC19] Peyre, G., and Cuturi, M., 2019. Computational Optimal Transport: With Applications to Data Science. Foundations and Trends in Machine Learning, Vol. 11, pp. 355-607.

[PoB94] Polymenakos, L. C., and Bertsekas, D. P., 1994. "Parallel Shortest Path Auction Algorithms," Parallel Computing, Vol. 20, pp. 1221-1247.

[SGS21] Swiechowski, M., Godlewski, K., Sawicki, B. and Mandziuk, J., 2021. "Monte Carlo Tree Search: A Review of Recent Modifications and Applications," arXiv preprint arXiv:2103.04931.

[San15] Santambrogio, F., 2015. Optimal Transport for Applied Mathematicians, Springer Intern. Publ.

[Sch16] Schmitzer, B., 2016. "A Sparse Multiscale Algorithm for Dense Optimal Transport," J. of Mathematical Imaging and Vision, Vol. 56, pp. 238-259.

[Sch19] Schmitzer, B., 2019. "Stabilized Sparse Scaling Algorithms for Entropy Regularized Transport Problems," SIAM Journal on Scientific Computing, Vol. 41, pp. A1443-A1481.

[SuB18] Sutton, R., and Barto, A. G., 2018. Reinforcement Learning, 2nd Ed., MIT Press, Cambridge, MA.

[TsB00] Tseng, P., and Bertsekas, D. P., 2000. "An ϵ-Relaxation Method for Separable Convex Cost Generalized Network Flow Problems," Mathematical Programming, Vol. 88, pp. 85-104.

[Vil09] Villani, C., 2009. Optimal Transport: Old and New, Springer, Berlin.

[Vil21] Villani, C., 2021. Topics in Optimal Transportation, American Mathematical Soc.

[WaD17] Walsh III, J. D., and Dieci, L., 2017. "General Auction Method for Real-Valued Optimal Transport," arXiv preprint arXiv:1705.06379.

[WaD19] Walsh III, J. D., and Dieci, L., 2019. "A Real-Valued Auction Algorithm for Optimal Transport," Statistical Analysis and Data Mining: The ASA Data Science Journal, 12(6), pp. 514-533.

[WaX12] Wang, J., and Xia, Y., 2012. "Fast Graph Construction Using Auction Algorithm," arXiv preprint arXiv:1210.4917.

[ZSP08] Zavlanos, M. M., Spesivtsev, L., and Pappas, G. J., 2008. "A Distributed Auction Algorithm for the Assignment Problem," Proc. 47th IEEE Conference on Decision and Control, pp. 1212-1217.