HW 2 Solutions

Yan Wang

November 30, 2015

Problem 1

a.

- $h(n) = min\{h_1(n), h_2(n)\}$ [3pts]
  - **- admissible:** *True.* Let n be any state in the graph, as $h(n)$ by definition must be equal to either $h_1(n)$ or $h_2(n)$, which is upper bounded by the actual cost $h^*(n)$, $h(n)$ is also upper bounded by $h^*(n)$. Q.E.D.
  - **- consistent:** *True.* Let n and m be any two adjacent states in the graph, from the condition that h_1 and h_2 are consistent, then we have $h(n) \leq h_1(n) \leq c(n,m) + h_1(m)$ and $h(n) \leq h_2(n) \leq c(n,m) + h_2(m)$, combining above two inequalities, we conclude that $h(n) \leq c(n,m) + min\{h_1(m), h_2(m)\} = c(n,m) + h(m)$ for each pair of adjacent states n and m. Q.E.D.
- $h(n) = wh_1(n) + (1-w)h_2(n),\ \text{where } 0 \leq w \leq 1$ [3pts]
  - **- admissible:** *True.* Let n be any state in the graph, from the condition that h_1 and h_2 are admissible and $w \in [0,1]$, then we have $h(n) \leq wh^*(n) + (1-w)h^*(n) = h^*(n)$ for each state n. Q.E.D.
  - **- consistent:** *True.* Let n and m be any two adjacent states in the graph, from the condition that h_1 and h_2 are consistent, then we have $h(n) \leq w(c(n,m) + h_1(m)) + (1-w)(c(n,m) + h_2(m))$. Rearranging the right side of the above inequality, we have $h(n) \leq c(n,m) + wh_1(m) + (1-w)h_2(m) = c(n,m) + h(m)$ for each pair of adjacent states n and m. Q.E.D.
- $h(n) = max\{h_1(n), h_2(n)\}$ [3pts]
  - **- admissible:** *True.* Let n be any state in the graph, as $h(n)$ by definition must be equal to either $h_1(n)$ or $h_2(n)$, which is upper bounded by the actual cost $h^*(n)$, $h(n)$ is also upper bounded by $h^*(n)$. Q.E.D.
  - **- consistent:** *True.* Let n and m be any two adjacent states in the graph, and as $h(n)$ must be one of $h_1(n)$ and $h_2(n)$, we can assume $h(n) = h_1(n)$ without loss of generality, then from the condition that h_1 and h_2 are consistent, we have $h(n) = h_1(n) \leq c(n,m) + h_1(m) \leq c(n,m) + max\{h_1(m), h_2(m)\} = c(n,m) + h(m)$ for each pair of adjacent states n and m. Q.E.D.

b. [6pts]

Without proof we make the following statement (*which is a lemma proven in the textbook and in class*): **on each step of the search algorithm, all paths from the start to the goal will cross/intersect with the open list**. Let's denote the open list as O. A direct corollary from above statement is: **if O is empty on some step, then there is no path from the start to the goal**.

- Short version
The algorithm is guaranteed to be optimal for $0 \leq w \leq 1$, since scaling $g(n)$ by a constant has no effect on the relative ordering of the chosen paths, but, if $w > 1$ then it is possible the $wh(n)$ will overestimate the distance to the goal, making the heuristic inadmissible. If $w \leq 1$, then it will reduce the estimate, but it is still guaranteed to underestimate the distance to the goal state.

- Long version
We assume that the heuristic function $h(s)$ is not trivial, i.e. $\exists n \in S$ s.t. $h(n) > 0$. and denote the actual cost from a state s to the goal as $h^*(s)$. Then based on the above statement, we propose and prove the following claims:

- **If $w \geq 2$, there exists a setting that the search algorithm returns a non-optimal path.**
  proof: **a)** First, let's consider the case $w > 2$. For this case, let's assume O contains only two states g_1 and g_2 on some step, s.t. $g(g_1) = 100$, $g(g_2) = 10$, and both correspond to the goal (w.r.t different paths), i.e. $h(g_1) = h^*(g_1) = h(g_2) = h^*(g_2) = 0$. Therefore, we have $f(g_1) = (2 - w) * 100 < (g_2) = (2 - w) * 10$, as $2 - w < 0$. According to the search strategy, the algorithm will pick g_1 and output the path of it, whereas the optimal one should be g_2. **b)** Then, let's consider the case $w = 2$. For this case, let's assume $O = \{g_1, g_2\}$, s.t. $g(g_1) = g(g_2)$, $h^*(g_1) = h(g_1) = 2$, and $h^*(g_2) = 5 > h(g_2) = 1$ (h is consistent). In this case, g_1 should be the optimal one, whereas the strategy will pick up g_2. Q.E.D.

- **If $w \in (1, 2)$, there exists a setting that the search algorithm returns a non-optimal path.**
  proof: assuming we are working with the optimal heuristic, i.e. $\forall s\ h(s) = h^*(s)$, as $w \in (1, 2)$, we have $\frac{2-w}{w} > 1$ and therefore the interval $\Big(g(s) + h^*(s), g(s) + \frac{w}{2-w}h(s)\Big)$ is not empty for all states. Let's now consider the following setting: on some step, O contains two states g and s, s.t. $h(g) = h^*(g) = 0$, $h^*(s) > 0$, and critically $g(g) \in \Big(g(s) + h^*(s), g(s) + \frac{w}{2-w}h(s)\Big)$. Apparently, state s corresponds to the optimal path. However, the algorithm will pick g according to the search strategy, as $g(g) < g(s) + \frac{w}{2-w}h(s) \Rightarrow (2 - w)g(g) + wh(g) < (2 - w)g(s) + wh(s)$. Q.E.D.

- **If $w \leq 1$, the search algorithm is guaranteed to output the optimal path for all settings.**
  proof: **a)** First, let's consider the case $w < 0$. In this case, assuming there exists a path from start to goal (otherwise trivially covered by previous lemma) and on some step the algorithm pick a state g which touches the

goal, i.e. $h(g) = h^*(g) = 0$. According to the search strategy, we have $\forall s \in O\backslash\{g\}\ g(g) \leq g(s) + \frac{w}{2-w}h(s)$. As $\frac{w}{2-w} < 0$ and $h^*(s) \geq h(s) \geq 0$, we have $g(g) \leq g(s) + \frac{w}{2-w}h(s) \leq g(s) + h^*(s)$. Hence, g is optimal. **b)** Then, let's consider the case $w \in [0, 1]$. Let's define a new heuristic $h'(s) = \frac{w}{2-w}h(s)$. Then h' is still admissible, as $\frac{2-w}{w} \leq 1$. And we observe that the search strategy is equivalent to using the objective $f'(n) = g(n) + h'(n)$, which is equivalent as conducting an admissible A^* search. Therefore the algorithm is guaranteed to output the optimal path. Q.E.D.

Combining above claims, we can conclude that: **by using an admissible heuristic, the algorithm is guaranteed to output the optimal path for all settings if and only if** $w \leq 1$.

For different w values, the search algorithm corresponds to existing methods:

w	f(n)	Algorithm
$w = 0$	$f(n) = 2g(n)$	Uninformed best-first search
$w = 1$	$f(n) = g(n) + h(n)$	A^* search
$w = 2$	$f(n) = 2h(n)$	Greedy best-first search

Problem 2

a. [2pts]

When the problem is about optimizing an objective function, which is **concave** [1] and **smooth** [2], then the randomness is not necessary, as the convergence to the maximum is assured for Hill Climbing. See the figure 1 for example.

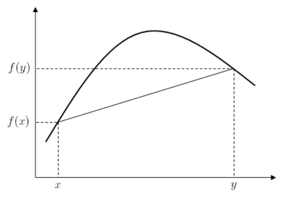

Figure 1: Concave function

b. [2pts]

When the problem is about optimizing an objective function, which has a large number of maximums whose objective values are close to each other, and is almost flat around them, then random guessing plays as well as the Simulated Annealing.

c. [2pts]

S.A. is useful when the objective function has multiple peaks (local maximum) which are different w.r.t. their objective values. See the figure 2 for example.

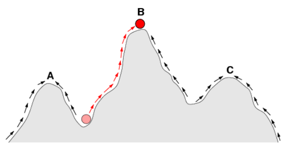

Figure 2: Function with multiple peaks

d. [2pts]

Keep and return the best (maximal) state seen so far among the sequence of states visited by the algorithm.

e. [2pts]

Divide the search domain into 2M equally large sub-domains and run an independent S.A. procedure on each sub-domain for a few steps. As every S.A. procedure is independent from the others, we can run these 2M S.A. procedures in parallel. When all procedures finish, we can output the best solution among all 2M returns. This can be implemented using the Map-Reduce framework [3].

Problem 3

a. [10pts]

We are going to prove the following claim:
If there is a path from start to goal, A^*_ϵ returns an ϵ-approx solution s i.e. $Cost(s) \leq (1+\epsilon)Cost(s^*)$ where s^* is the optimal solution; Otherwise, A^* stops at some step s.t. $O = \emptyset$.
proof: remember the claim we used in problem 1b: the open list O always intersect with all the paths from start to goal if there exists one. We are going to reuse this claim for this proof. Assuming there exists a path from start to goal (otherwise trivially covered by the claim), then A^*_ϵ will eventually output one at some step. Let's denote the last state expanded by A^*_ϵ as s and the state within the intersection of the optimal path and the open list O as s^*. As s touches the goal, its heuristic is zero. According to the selection strategy, we then have the following inequalities: $g(s) \leq (1+\epsilon)min_{n\in O}\{g(n) + h_A(n)\} \leq g(s^*) + h_A(s^*)$. As h_A is admissible, $g(s^*) + h_A(s^*)$ is always upper bounded by the actual cost of s^*, i.e. the optimal cost. Therefore, $g(s) = Cost(s) =\leq (1+\epsilon)Cost(s^*)$. Q.E.D.

b. [5pts]

The second heuristic function h_N can be used to break ties for each selection from the open list. As the states in the open list satisfying the selection criteria could be multiple, we can use the second heuristic function (might be non-admissible) h_N to pick the one with the minimal heuristic value to proceed on. By decreasing ϵ, the candidate set will shrink, and the tie-breaker h_N will play less on the selection; in the other way, by increasing ϵ, the candidate set will expand, and therefore h_N will play more to break the tie.

Problem 4

a. [3pts]

See figure 3 for solution.

b. [3pts]

See figure 4 for solution. The main idea here is: **when you observe a good-enough return from one branch, you can ignore the rest**! For example, in the figure 4, on state $(3, 2)$, as its branch of $(3, 1)$ returns -1 which is already optimal for agent B, there is no need to further investigate into the branch of $(3, 4)$, which incurs an infinite loop to the standard minimax algorithm. Same argument can be made on state $(1, 3)$.

c. [3pts]

The minimax algorithm recursively performs the following actions in a game tree: **a)** compute game values for all its child nodes; **b)** on each min(max) node, it returns the min(max) of the game values of its child nodes; **c)** if it is a terminal node, it returns the game value of that ending. The standard minimax will fail because the recursion will fall into the endless loop, marked by state $(1, 4)$ and state $(2, 4)$.
We can modify the minimax algorithm in the following way: during the search, when exploring on a node, we firstly look at the game values returned from the explored children of it at the moment, if we have a good enough value then we can stop (i.e. pruning) the recursion to its unexplored children and return that value. Otherwise we look for the previous occurrences of the state in the current search path: if there is no occurrence before it, we can choose any child of it to explore; if there exists a occurrence before and its children haven't been fully explored yet, we can choose any unexplored child to move on. Otherwise we stop the entire procedure and return nothing, as there exists some inherent infinite loop of the game that we can't skip by using the good-enough pruning.

d. [1pts]

We can prove the claim by induction on the number of cells:

- **base:** When $n = 3$, B wins as A starts moving to cell 2 and then B can directly jump into the left end cell; when $n = 4$, as the game tree in figure 4 shows, A wins following the optimal path.
- **assumption:** for all $t \leq k$, where $k \in \mathbb{Z}$ and $k \geq 2$, A wins for $n = 2t$ and B wins for $n = 2t - 1$.
- **induction:** now, let's consider the next step, i.e. $k + 1$. There are two cases to cover, i.e. $n = 2k+1$ and $n = 2k+2$, and we start with $n = 2k+1$:
  - **$n = 2k + 1$:** The optimal strategy for B is **always moving left**. The only situation where B cannot move left is the state $(1, 2)$ with B's turn. Thus B has to move to cell 3, A then moves to cell 2, and finally B can move to cell 1. A key observation is that: **if B uses**

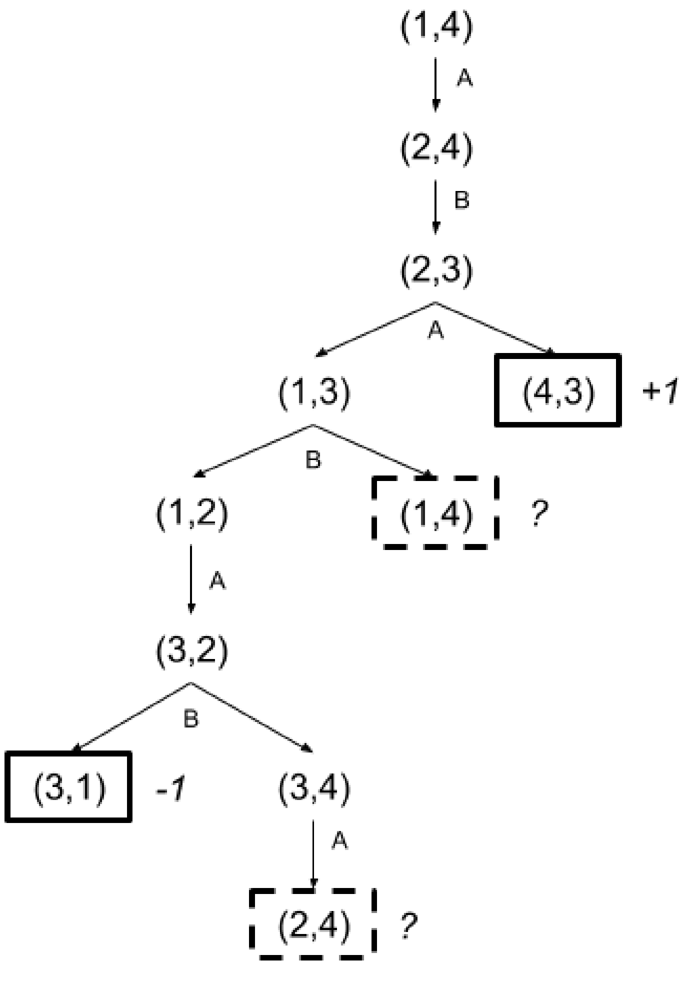

Figure 3: Game tree without backed-up minimax values

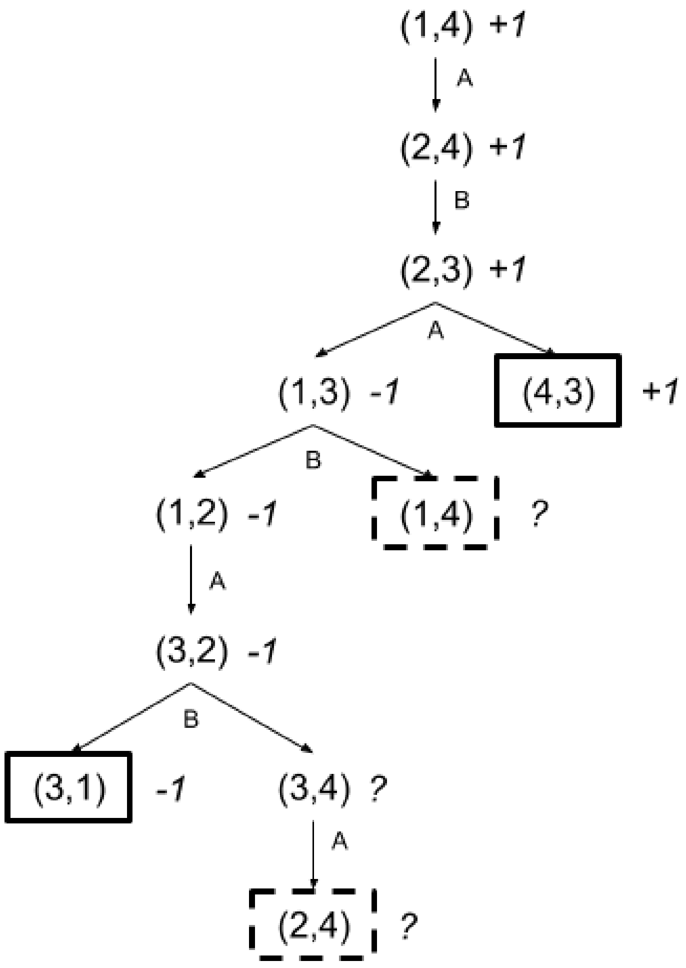

Figure 4: Game tree with backed-up minimax values

this strategy, assuming the first state where A and B meet is $(k', k'+1)$, then $k' \leq k+1$. **a)** If $k' \leq k-1$, no matter who takes the turn, B will be in an advantageous situation as it has less cells left towards cell 1 than the number of cells left towards cell n for A. Following a simple argument, it can be shown that B will win in this situation. **b)** If $k' = k+1$, it must be B's turn to move, as based on simple counts B moved $k-1$ steps and A moved k steps before they meet. Therefore B can jump over A and move to cell k which results in the state $(k+1, k)$ with A's turn. Then B has $k-1$ cells left towards cell 1, while A has k cells left towards cell n, which brings B the advantage over A. Therefore B will win in this situation. **c)** $\mathbf{k' \neq k}$, as based on simple counts B moved k steps and A moved $k-1$ steps before they meet, which is **impossible**, as A starts moving and they make moves in turn. Combining all above three cases, we conclude that B **will win**.

- **$n = 2k+2$:** This case can be easily reduced to the $n = 2k+1$ case which we've discussed above through considering the situation after the first move by A: for the first move, A can only move to cell 2. After then, if we discard cell 1 and instead take cell 2 as the left end, then the game is reduced to the $n = 2k+1$ case where B starts moving, i.e. A and B interchanging their roles in the reduced game. The interchange of roles doesn't affect analysis, thus following the same arguments in case $n = 2k+1$, we conclude that A **will win**.

Problem 5

a. [2pts]

You can apply any DFS-based algorithm here, but not BFS, as it consumes too much memory. Choosing between fill-in by letter or by word is actually making a trade-off between the search depth and branching factor. **a)** If choosing to fill in blanks by letter, the branching factor is fixed as 26, while the depth is equivalent to the number of open squares, which is usually very large. **b)** If choosing to fill in blanks by word, the branching factor varies for each node and depends on the number of open squares (i.e. length) to be filled on the node, i.e. the number of words with the same length. Though the branching factor might increase, the depth can be remarkably decreased down to the number of words in the grid. So, for the cases where there are many long words to be filled, filling the blanks by word should be preferred, as a long-word node usually have small branching factor and the search depth can be decreased to a much lower level compared to the filling-by-letter choice. You may worry about the time on searching through the dictionary for a valid word satisfying certain prefix/suffix constraints. This should not be a concern as we can apply efficient data structures on dictionary representation, e.g. prefix tree[4] and suffix tree[5].

b. [2pts]

If we choose letters, a variable is each box in the puzzle and the constraints are the variables must combine to make a word in the dictionary. If strings of

boxes are variables, the domain is the set of words from the dictionary with the constraint that the intersection of two words must have the same letter in the intersecting box.
CSPs are solved using general search algorithms. An appropriate algorithm for the puzzle problem is DFS (from the previous question). Therefore, defining the variables as words is probably a better choice.

c. [1pts]

The CSP formulation is better, because arc-consistency algorithms (such as AC-3) can be used to reduce the domains of the variables before performing the search.

Problem 7

c. [5pts]

- **Enumeration** takes time $O(n2^{n-1})$, as there are 2^{n-1} possible instances and computing each one costs $O(n)$ products.
- **Variable Elimination** with the order $V_2 \to V_3 \cdots \to V_{n-1}$ takes linear time $O(n)$, as eliminating each variable V_i only involves two factors $P(V_{i+1}|V_i)$ and $P(V_1, V_i)$ which is the result of previous eliminations, and gives a new factor $P(V_1, V_{i+1})$ as the result.

Problem 8

a. [5pts]

proof: let's consider the factorization of the joint distribution w.r.t. its Bayesian network representation, i.e.

$$P(V) = \prod_{v_i \in V} P(v_i|Parents(v_i))$$

Then we can group all local factors with X in it into one single factor G and all the rest into another factor F, i.e.

$$P(V) = G(X, MB(X))F(V \backslash X)$$

where $G(X, MB(X))$ has the following formula:

$$G(X, MB(X)) = P(X|Parents(X)) \prod_{Y_i} P(Y_i|Parents(Y_i))$$

After regrouping the factorization, the conditional distribution $P(X|MB(X))$ can be written as follows:

$$P(X|MB(X)) = \frac{P(X, MB(X))}{\sum_X P(X, MB(X))} = \frac{\sum\limits_{V \backslash (MB(X) \cup X)} G(X, MB(X))F(V \backslash X)}{\sum\limits_X \sum\limits_{V \backslash (MB(X) \cup X)} G(X, MB(X))F(V \backslash X)}$$

As the set $V\backslash(MB(X)\cup X)$ doesn't intersect with $MB(X)\cup X$ and is contained in $V\backslash X$, the summation over variable set $V\backslash(MB(X)\cup X)$ can be moved to be combined with the factor $F(V\backslash X)$. Let's define the following summation term: $S(MB(X)) = \sum\limits_{V\backslash(MB(X)\cup X)} F(V\backslash X)$, then the conditional distribution can be simplified as follows,

$$P(X|MB(X)) = \frac{G(X,MB(X))S(MB(X))}{\sum\limits_{X} G(X,MB(X)) * S(MB(X))}$$

and it can be further simplified as both the dividend and divisor contain the common term $S(MB(X))$, i.e.

$$P(X|MB(X)) = \frac{G(X,MB(X))}{\sum\limits_{X} G(X,MB(X))}$$

If we denote the normalization term $\sum\limits_{X} G(X,MB(X))$ as $\frac{1}{\alpha}$ and expand the formula of $G(X,MB(X))$, then we can get the following equation:

$$P(X|MB(X)) = P(X|Parents(X))\prod_{Y_i} P(Y_i|Parents(Y_i))$$

Q.E.D.

b. [5pts]

We use MCMC to sample a sequence of $(Cloudy, Rain)$ pairs and then compute the empirical marginal distribution for variable $Rain$ from the sequence. The details of MCMC algorithm can be found there[6]. Conditioned on that $(Sprinkler = true,\ WetGrass = true)$, there are 4 possible states in the MCMC procedure:

Cloudy	Rain	Sprinkler	WetGrass
True	*True*	*True*	*True*
True	*False*	*True*	*True*
False	*True*	*True*	*True*
False	*False*	*True*	*True*

c. [5pts]

The transition matrix of the MCMC algorithm conditioned on $(\bar{S} = true, \bar{W} = true)$ can be computed using the following formula:

$$T(C,R \rightarrow C',R') = \begin{cases} \frac{1}{2} * P(C'|R,\bar{S},\bar{W}) & \text{if } C' \neq C \text{ and } R = R' \\ \frac{1}{2} * P(R'|C,\bar{S},\bar{W}) & \text{if } C' = C \text{ and } R \neq R' \\ \frac{1}{2} * P(C|R,\bar{S},\bar{W}) + \frac{1}{2} * P(R|C,\bar{S},\bar{W}) & \text{if } C' = C \text{ and } R = R' \\ 0 & \text{if } C' \neq C \text{ and } R \neq R' \end{cases}$$

where $C, C' \in \{true, false\}$ and $R, R' \in \{true, false\}$. Therefore the matrix is:

$$\begin{array}{c|cccc} & C(t)R(t) & C(t)R(f) & C(f)R(t) & C(f)R(f) \\ \hline C(t)R(t) & \frac{1}{2}*\frac{4}{9}+\frac{1}{2}*\frac{22}{27} & \frac{1}{2}*\frac{5}{27} & \frac{1}{2}*\frac{5}{9} & 0 \\ C(t)R(f) & \frac{1}{2}*\frac{22}{27} & \frac{1}{2}*\frac{5}{27}+\frac{1}{2}*\frac{1}{21} & 0 & \frac{1}{2}*\frac{20}{21} \\ C(f)R(t) & \frac{1}{2}*\frac{4}{9} & 0 & \frac{1}{2}*\frac{5}{9}+\frac{1}{2}*\frac{11}{51} & \frac{1}{2}*\frac{40}{51} \\ C(f)R(f) & 0 & \frac{1}{2}*\frac{1}{21} & \frac{1}{2}*\frac{11}{51} & \frac{1}{2}*\frac{40}{51}+\frac{1}{2}*\frac{20}{21} \end{array}$$

References

[1] Wikipedia, “Concave function — wikipedia, the free encyclopedia,” 2015. [Online; accessed 22-November-2015].

[2] Wikipedia, “Smoothness — wikipedia, the free encyclopedia,” 2015. [Online; accessed 22-November-2015].

[3] Wikipedia, “Mapreduce — wikipedia, the free encyclopedia,” 2015. [Online; accessed 22-November-2015].

[4] Wikipedia, “Trie — wikipedia, the free encyclopedia,” 2015. [Online; accessed 22-November-2015].

[5] Wikipedia, “Suffix tree — wikipedia, the free encyclopedia,” 2015. [Online; accessed 22-November-2015].

[6] Wikipedia, “Markov chain monte carlo — wikipedia, the free encyclopedia,” 2015. [Online; accessed 23-November-2015].